学習
BDSC・α-C

御手洗　毅・理英　著

工学図書株式会社

学 習
BDS C・α-C

御手洗　毅・理英 著

工学図書株式会社

学　習
BDS C・α-C

御手洗　毅・理英 著

工学図書株式会社

ま え が き

C言語は現在，パソコンの機種を越えた共通の言語として使われるようになり，学習する方も非常に多くなりました．中でも，BDS C，α-Cは8ビットパソコン上で動作する本格的なCコンパイラとして有名で，安価で操作が簡単なことから，初心者も気軽に学習をはじめる事ができます．

ただ，残念なことに，C言語はシステム記述言語と呼ばれるように，OSなど高度なプログラム開発を目的とした言語であるため，学習は簡単ではありません．さらに，BDS Cは8ビットのC言語として最もポピュラーなものであるにもかかわらず，標準的なC言語から一部機能の削除されたサブセット版であり，標準Cの参考書では学習しにくい場合があることも事実です．

そのため，本書ではC言語そのものの解説は最小限に留め，BDS C，α-Cを学習する上で判りにくいポイント，重要なポイントが把握できるようにサンプルプログラムを中心に解説しました．初心者の方には最適の教科書になると思われます．

さらに，応用編としてBDS C，α-Cにおける日本語処理，オーバーレイ技法などを取り上げ，初心者だけでなく，本格的にBDS Cを使っている方も十分役立つ内容となっています．

是非，本書を活用されることを期待しています．

なお，本書では次のバージョンを用いています．

BDS C　　バージョン1.50a
α-C　　バージョン1.51

昭和62年1月

御手洗　毅

目　次

第1章
BDS C入門

BASICでは，パソコンの電源を入れてBASICを起動させ，そのあとプログラムが完成するまでフロッピーディスクを抜いたままで作業を行なうことができます．これは，BASICインタプリンタ本体がパソコンのメモリ上に常駐しており，その上で，プログラムの作成，変更，実行といった一連のプログラム開発作業がすべて行なえるようになっているからです。

BDS CはCP/Mのもとで動作しますが，BDS Cに限らず，すべてのCP/M用のソフトウェアはディスクが基本になっており，フロッピーディスクがなければ一切の作業を行なうことができません．エディタ，言語(コンパイラ）などは異なるプログラムとして独立してディスク内に記録され，必要に応じて実行されるわけです．これは大変不便なようですが，メモリに常駐している部分が小さいため，大きなマシン語プログラムを動作させることができるという特徴を持っています．ちょっとしたプログラムを作るだけなら，BASICで十分という場合もありますが，本格的なプログラムを動作させたい，作成したいという場合には，CP/Mを用いることは常識であり，中でもBDS Cは本格的な8ビット用のソフトウェアを作るのに向いています．

1-1 BDS Cのセットアップ

(1) BDS C, α-C利用の手順

BDS C, α-Cを用いたプログラム作成の概要について説明します.

まず, C言語のソースファイルはテキストエディタを用いて作成します. CP/Mを購入した際にED.COMというエディタが付属していますのでそれを用いることもできますが, EDは1行ごとに表示させながら行単位で書いていくラインエディターですので, 一般には画面にプログラムを表示させ, カーソルにより1文字ずつ編集するスクリーンエディタを利用する方が便利です.

CP/M-80用には米マイクロプロ社のワードマスターやワードスターが有名ですが, C言語のソースプログラムをディスクファイルとして作成できるものであれば何でも構いません. しかし, エディタが使えなければBDS Cは使うことができませんので, その操作に慣れることがC言語に慣れる第一歩と言っても過言ではないでしょう.

次に, 作成したソースファイルをBDS Cあるいはα-Cコンパイラによってコンパイルします. コンパイラ本体は両者とも同じファイル名で,

```
cc.com
cc2.com
```

という2つから成り, これらがオーバーレイ(チェイン)してコンパイル作業を行ないます. ここで重要なのは, これらのプログラムで処理された出力はそのままCP/M上で実行できるわけではなく, 一旦リロケータブルなバイナリ形式のファイルとしてディスク上に作成されることです. これは複数のソースファイルから1つの実行プログラムを作成したり, すでにBDS Cに備わっている別のプログラムモジュール(標準関数など)と組み合せる際に

必要だからです．

実行ファイルを作成するためには，リンカーを用います．リンカーは

```
clink.com
```

という専用プログラムが用意されています．これを用いると，実行形式のCOMファイルが得られ，作成したプログラムを実際に動作させてみることができるわけです．cc，cc2，clinkなどの基本ソフトウェアはまず，マスターディスクからピックアップし，作業用ディスクにコピーしておくことが必要ですが，使用ディスクのタイプによって要領がかなり異なってきます．1MB以上のディスクを使う場合には1基でも余裕がありますので，滅多に使わないプログラムもコピーしておくことが可能ですが，320KB程度のドライブの場合にはどうしても最低限必要なファイルだけをまとめておき，それ以外のファイルが必要になった時にはディスクを抜き差しして使うことになります．320KBドライブの場合，1基でも利用は可能ですが，大きなプログラムを作るとすぐディスクが一杯になってしまいますので，実用的には2基以上必要です．

〈最低限必要なファイル　(BDS C，α-C共通)〉

```
cc.com
cc2.com
clink.com
deff.crl
deff2.crl
c.ccc
bdscio.h
```

〈必要に応じて利用するファイル　(BDS C，α-C共通)〉

```
clib.com        (Cライブラリアン)
```

```
dio.h        (ダイレクトコンソールIOパッケージ)
dio.crl      (ダイレクトコンソールIOパッケージ)
```

<必要に応じて利用するファイル (BDS Cのみ)>

```
float.crl    (浮動小数点演算パッケージ)
long.crl     (倍精度整数演算パッケージ)
casm.com     (アセンブラパッケージ)
casm.sub     (アセンブラパッケージ)
cdb.com      (シンボリックデバッガー)
cdb2.ovl     (シンボリックデバッガー)
l2.com       (cdb用リンカー)
```

【注意】 dio, float, long, cdb, l2などはマスターディスクにソースリストの形で格納されていますので，マニュアルに従って，あらかじめ処理する必要があります．また，casmを用いる際にはCP/Mの標準ユーティリティASM，DDTが必要になります．

<最低限必要なCP/Mユーティリティ>

```
PIP.COM
STAT.COM
SUBMIT.COM
XSUB.COM
ED.COM       (あるいは他のエディタ)
```

その他，ディスクフォーマット・ディスクバックアップ用のユーティリティなど．

(2) ソースファイルの作成

必要なファイルを格納した作業用のディスクを作ると，プログラムの作成を始めることができます．

まず，最初にC言語のソースファイルを作成します．ここでは最も簡単な

プログラムとして，画面に文字を出力するプログラムを作成してみます.

リスト1-1-1 hello.c

```
#include <bdscio.h>

main()
{
        printf ("hello, world¥n");
}
```

この内容のファイルをCP/M付属のエディター，EDを用いて作ってみましょう. 下線部が実際に手でタイプした部分です.

操作例1-1-2 hello.cの入力

```
A>ed hello.c

NEW FILE
     : *i
    1:  #include <bdscio.h>
    2:
    3:  main()
    4:  {
    5:          printf ("hello, world¥n");
    6:  }
    7:  ^Z
     : *e

A>
```

EDには多くのコマンドがあり，ファイルの作成，修正，編集などが行なえるようになっていますが，詳細は省略します.

さて，プログラムはたった5行ですが，これはC言語の基本的なマニュアルである「プログラミング言語C」* という本の最初にでてくるサンプルプロ

＊巻末の参考文献参照.

グラムで，おそらく現在C言語を使っている多くの方が1度は実際にタイプしてみた例の1つだと思います．

一目でわかるように，Cはその多くの部分を小文字で記述します．見た目に美しく，かつ{，}などの記号を用いることでタイプ量が少なくなっているのもその特徴の1つになっています．また，強制ではありませんが，対応関係をはっきりさせるために，インデント（字下げ）を行ないます．

さて，実際のプログラムですが，

```
#include <bdscio.h>
```

という第1行はBDS Cのプログラムを書く上での決まり文句です．初心者のうちは，第1行は必ずこのように書くと考えて構わないでしょう．実際には，ディスクから

```
bdscio.h
```

というファイルを読み込み，この行と入れ換える動作を行ないます．

次の行から実際のプログラムがスタートします．

```
main( )
```

mainはこの関数の名前です．関数というのは，C言語におけるプログラムの単位で，1つのサブルーチンと考えれば良く，(　)の中には通常引数を列挙します．たまたまここでは引数はありませんが，カッコを省略することはできません．そして実際のプログラム本体は次のカーリーブレイス記号{から，最後の}までとなります．{ }はプログラムの中にも登場しますので，それらの対応関係が正しくなるように関数単位で{と}の数を合わせる必要があります．

なお，関数名はキーワード（ifやforなど)・変数名とダブらなければ自由に決めることができるのですが，mainという関数名だけは，固定（変更は可能）で，プログラムそのものがスタートした時，必ずこのmainという関数

から実行が開始される決まりになっています．いくつも他の関数が定義されていて，mainが最後の方にあったとしても実行は必ずmainから始まるわけです．言い方を換えると，必ず関数mainは必要だということです．

```
printf ("hello,world¥n") ;
```

この1行が実際に動作するプログラムの本体です．“hello, world”という文字列を画面に表示するわけですが，ここで用いている関数printfはBASICのPRINT文などと異なり，C言語の機能としてあらかじめ用意されているわけではなく，あくまで1つの関数として扱われ，ユーザーが作成した関数との区別はありません．つまり，単にBDS Cの作成者がユーザーの便利のために作っておいたサブルーチンの1つなのです．

BASICではPRINT文などのあらかじめ備わっている機能と，ユーザーが作成したサブルーチンは完全に異なるものであり，新たにステートメント(文)を追加・拡張したりすることができません．

さて，このように関数を利用する場合には，まず関数名を書き，( )の中にその関数への引数を記述します．この場合は“ ”で囲んだ文字列が引数ということになっているわけです．なお，文字列の最後の¥nは改行の指定です．文字として書くことができない制御コードなどは，¥マークと文字を組み合せて表現することになっています．BASICで，

```
PRINT "hello"          と,
PRINT "hello";
```

の違いはお判りですね．BASICと逆に，改行する場合にこのような付加記号が必要になってくるのです．

なお，文の終りには，必ずセミコロン（；）を付けます．

それでは，実際のコンパイル作業を行なってみましょう．このプログラムは

```
hello.c
```

というファイル名で，ドライブAに作成しましたから，操作例1－1－3のようにコンパイルし，実行してみてください．

操作例1－1－3　hello.cのコンパイル

```
A>cc hello ←―――― hello.cのコンパイル．hello.crlというファイルが作成される．
BD Software C Compiler v1.50a (part I)
  35K elbowroom
BD Software C Compiler v1.50 (part II)
  32K to spare

A>clink hello ←―――― hello.crlからhello.comという実行ファイルを作成する．
BD Software C Linker   v1.50
Linkage complete
  43K left over

A>hello ←―――― helloの実行．
hello, world

A>
```

正しく実行できたでしょうか．これらのコンパイル，リンクの流れを図1－1－4に図示しておきます．

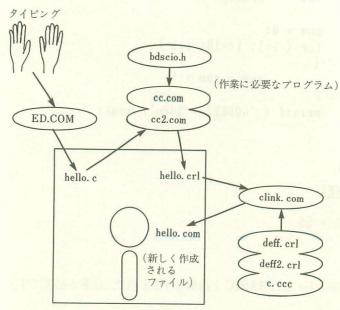



図1-1-4　BDS Cのコンパイル作業

1-2　初心者のためのサンプルプログラム

すべてのコンピュータ言語に共通して言えることですが，上達への早道は，簡単な短いプログラムを数多く作ってみることです．本項では，学習に適したサンプルプログラムを解説とともに示しますので，実際にタイプし，実行してみてください．

(1)　1から10までを加算して表示するプログラム

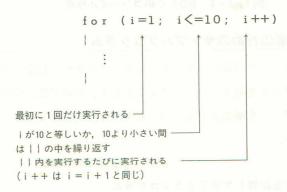
リスト1-2-1　goukei.cと実行例

```
#include        <bdscio.h>

main()

{
```

```
        int     i,sum;

        sum = 0;
        for ( i=1; i<=10; i++ )
        {
                sum = sum + i;
        }
        printf ( "GOUKEI = %d¥n", sum);
}
```

〈実行例〉

```
A>GOUKEI

GOUKEI = 55

A>
```

C言語のfor文はBASICとは異なりますので，注意が必要です．

条件は制御変数iと終値との比較だけでなく，iと全く関係のない比較文を書くこともできます．また，単純増加，減少だけでなく，i++のかわりにi=i*2などとすることもできますので，はるかに用途が広くなっています．また，条件のチェックはループの実行前に行なわれます．従って，最初から条件が成立しなければループ内部は一度も実行されません．

(2)　100までの素数を表示するプログラム

リスト1-2-2　sosuu.cと実行例

```
#include        <bdscio.h>

main()
{
        int     i,s;
        char    f;

        for (s=2; s<=100; s++)
        {
                f=0;
                for (i=2; i<=s-1; i++)
                {
                        if (s%i==0)
                        {
                                f=1;
                                break;
                        }
                }
                if (f==0)       printf ("%d¥t",s);
        }
}
```

〈実行例〉

```
A>SOSUU

2       3       5       7       11      13      17      19      23      29
31      37      41      43      47      53      59      61      67      71
73      79      83      89      97

A>
```

%は剰余演算子で，s%iでsをiで割った余りを求めます．

リスト1-2-2はある数に対し，2からその数より1小さい数までしらみつぶしに割り切れるかどうかを調べ，割り切れなかった場合を素数とします．この例では，割り切れた場合にフラグfを1にし，それ以降チェックを省略し

ます．このようなループの中途強制中断にはbreakを用います．

ループの外でフラグfを見て，0であれば割り切れる数がなかったのですから，素数として画面に表示します．

¥tはタブコードで，TABキーを押したのと同様，数字と数字の間にスペースをあけます．¥nなどと同じ文字列中の特殊文字です．

(3) 任意の2つの数字を手で入力し，その加算結果を表示するプログラム

リスト1-2-3　nyuryoku. cと実行例

```
#include        <bdscio.h>

main()
{
        int     d1,d2;

        printf ("input=");
        scanf ("%d,%d",&d1,&d2);
        printf ("%d+%d=%d¥n",d1,d2,d1+d2);
}
```

〈実行例〉

```
A>NYURYOKU

input=5,10
5+10=15

A>
```

scanfはBASICのINPUT文に相当します．しかし,そう頻繁に用いられる関数ではありません.ここで,引数を指定する際に変数名の頭に＆記号を付けていることに注目してください．C言語では，関数の戻り値としてただ1つの値しか返すことができません．2つ以上の値を返さねばならない場合，＆記号

をつけて変数の物理的なアドレスを渡し，そのアドレス位置に書き込ませることによって，目的を達します．C言語ではアセンブラと同様，このようなアドレスをしばしば用います．他の高級言語には余り例がなく，C言語の特徴でもあります．２つの変数の値を入れ換えたりする場合にも引数を変数のアドレスにしなければ，実現できません．

(4) 手で入力した文字をそのまま画面に表示するプログラム

リスト1-2-4　typeout.cと実行例

```
#include        <bdscio.h>
#define         EOF     -1

main()
{
        int     c;
        while (!((c=getchar())==EOF)) putchar(c);
}
```

〈実行例〉

```
A>typeout

AABBCCDDEEFFGGHHIIJJKKLLMMNN

^Z

A>
```

このプログラムはキーボードからの１文字入力関数 getcher と出力関数 putchar の使用例ですが，getchar でその値を格納している変数 c が int（整数）型であることに注意してください．ファイル終了コードと同じ 0x1a（コントロールＺ）をキーボードから入力すると，入力の終了と判定し，getchar は－１を返しますが char(文字)型の変数は０から255までの値しか取れないため，－１という値との比較が成立しなくなります．

(5) 棒グラフの表示

リスト1-2-5　bargraph.cと実行例

```
#include        <bdscio.h>

main()

{
        int     d1,d2;

        printf ("data1->");
        scanf ("%d",&d1);           /* input data1 */
        printf ("data2->");
        scanf ("%d",&d2);           /* input data2 */

        bar (d1);
        bar (d2);
}

bar(n)                              /* output bar & data */
        int     n;
{
        int     i;
        for (i=1; i<=n; i++)    printf ("*");
        printf (" %d¥n",n);
}
```

〈実行例〉

```
A>bargraph

data1->15
data2->20
*************** 15
******************** 20

A>
```

アスタリスク(＊)を指定した数だけ画面に出力し，棒グラフを作るプログラムです．

ここで紹介したプログラムはBASICなどでも良く作成されるものですが，実際のプログラミング作業はBASICに比べ，難しくなります．しかしながら，その動作速度はさすがC言語と言わしめるものがあります．

これらはすべて，ディスクのファイルアクセス処理のない簡単なサンプルです．是非，すべてを理解して欲しいと思います．

第 2 章 Cのプログラムスタイル

第１章では全くの初心者の方がプログラムを作成する場合のサンプルをご紹介しました．Ｃ言語は非常にシンプルですが，細かい部分ではどうしてもテクニックを必要としますから，それが初心者の頭を悩ますところでもあります．

本章では，練習のためのサンプルプログラムではなく，本格的にBDS Ｃを使っていくために必要な部分を学習します．ざっと読み流しても構いませんが，判らないままにはせず，自分のものになるまで理解してください．実際にプログラムを作成する場合，大きな力になるはずです．

2-1 ポインタの使い方

ポインタは小型で高速なオブジェクトを出力するC言語におけるスーパースターであり，非常に特徴的な機能です．

この機能のおかげでシステム記述言語として，ハードウェアに密着したソフトウェアやOS，言語などが作成しやすいと言われているわけです．しかし，その利用は必ずしも簡単とは言えません．

2-2，2-3節で述べるコマンドラインの受け取り方や文字列の処理などでは，ポインタは使わないわけにはいかないのですが，ポインタは初心者程，無理に使おうとする傾向があり，使わなくても良い場合まで使いがちです．ポインタを利用するプログラムの多くは配列を用いて書き換えることが可能で，その方がプログラムの理解もたやすくなります．

ポインタの使い方に慣れてくると，ポインタを使うことでプログラムが小さく，作成も簡単になる場合と，複雑で混乱を招きやすくなる場合との区別がつくようになってきます．ポインタ，ポインタと呪文のように唱えるより，プログラムを早く，正確に作ることを最初は優先すべきだと思います．ただし，ポインタは重要な概念です．無理にポインタを使う事には反対ですが，ポインタを知らずに済ますことはできません．

(1) 文字列に対するポインタの使い方

さて，ポインタの具体的な使い方を説明するために，ポインタが最も多用される場合として，文字列の処理を扱ってみます．標準関数にstrlenという文字列の長さをカウントする関数がありますが，これと同じものを配列を用いて実現した例をリスト2-1-1に，ポインタを用いたものをリスト2-1-2に作成してみました．説明のため，関数名を前者はstrlen1，後者はstrlen2としていますが，動作は全く同じです．

なお，文字列については2-4節で詳細に触れています．

リスト2-1-1　strlen1.cと実行例

```
main()
{
        printf ("length=%d",strlen1("HELLO"));
}

strlen1(s)
        char    s[];
{
        int     i;
        i=0;
        while (s[i]) i++;
        return (i);
}
```

〈実行例〉

```
A>strlen1

length=5

A>
```

リスト2-1-2　strlen2.cと実行例

```
main()
{
        printf ("length=%d",strlen2("HELLO"));
}

strlen2(s)
        char    *s;
{
        int     i;
        i=0;
        while (*s++)    i++;
        return (i);
}
```

〈実行例〉

```
A>strlen2

length=5

A>
```

関数strlen1，strlen2によって，“HELLO”という文字列の長さを求めます．Ｃ言語の配列では，配列の０番めの要素よりメモリの下位番地（０に近い番地）から順に格納されています．

配列の０番めの要素	s［0］	（下位番地）
1	s［1］	
2	s［2］	
3	s［3］	
⋮	⋮	（上位番地）

Ｃ言語では必ずこのような配列が並ぶように定められているため，ポインタを使うことができるわけです．

この両者でまず異なるのは，引数ｓの受け方です．strlen1では文字列をchar（文字）型変数配列とみなし，ｓをその配列アドレス(が格納されている変数）として宣言していますが，strlen2ではポインタ（アドレスを内容とする変数）として宣言しています．従って，プログラム中でもstrlen2ではsの内容をダイナミックに変化させているのに対し，strlen1では定数として扱い，ｓの値は変化させていません．

利用する変数の数，アルゴリズム，行数も同じですし，動作も全く同じです．BASICに慣れた方なら，strlen1は問題無く理解できると思います．その上で両者を対応させ，異なる点を突き詰めていくと，ポインタの概念もお分りいただけるでしょう．

さて，プログラム中で実際に異なるのは，whileのある行だけです．プログ

ラムの考え方は全く同じですから，(　)の中にある，

```
s[i]
*s++
```

が，内容としては基本的に同じものでなければなりません．

ここで，s[i]は配列ですから，sという配列のi番めの要素です（実際にはsという変数の内容を先頭アドレスとする文字配列のi番めの要素）．最初にwhileの(　)の中が評価される時には，iは0ですから文字列の最初の文字を表します．while文は(　)の中が0でない時そのすぐ後の文を繰り返しますから，もし，そこが文字列の終端文字(0)でなければ，iをインクリメントします(i++)．

この部分がstrlen2では *s++ となっています．この場合の*は演算子で，変数sの内容をアドレスとする内容を読み出します．++はインクリメントの演算子で，読み出し操作と同時にsの内容を1増加させます．*は++よりも優先順位が低いので，*sの値（つまり文字列の中身そのもの）ではなく，変数sの内容（アドレス）を増加させるのだということに注意してください．つまり，*sで内容を参照すると同時に，

```
s=s+1;
```

を実行するのと同じです．従って，i=0の時，*s++も文字列の第1文字めを表します．

もし，*sの値を増やすのなら，(*s)++とし，優先順位を変更するための(　)がなければなりません．

さて，第1文字が0でなければ，次はiが1になります．従って，s[i]は，s[1]ですから，配列の2番めの要素を表します．同様strlen2では，sの値が1回インクリメントされていますから，*s++で2文字めの内容を示し，sの値をさらにインクリメントすることになります．

このように，表現は全く異なるものの，動作としては全く同じ関数を配列，

ポインタという2つの考え方を用いて作ることができるわけです．

念のため，ここで気になるバイト数ですが，関数の部分だけでstrlen 1 は62バイト，strlen 2は60バイトとなりました．速度は比較していませんが，大きな差はないものと思われます．

(2) 整数に対するポインタの使いかた

intなどの整数に対しては，charと比べるとポインタの利用はやや少なくなります．これは，もともと文字列がC言語ではchar型の変数配列として定義されているため，文字列を扱う場合にはポインタが避けて通れないのに対し，整数に対しては使わなくてもプログラムが作れるからです．

なお，整数配列も文字配列と同様，

配列の0番めの要素	i [0]	(下位番地)
1	i [1]	
2	i [2]	
3	i [3]	
⋮	⋮	(上位番地)

のように並びます．ただし，1つの要素は2バイトずつになります．

サンプルとして，バブルソートのプログラムを作ってみましょう．バブルソートはより効率の良いソート方法がいくつも考案されていますが，アルゴリズムが単純なため現在でも良く用いられます．

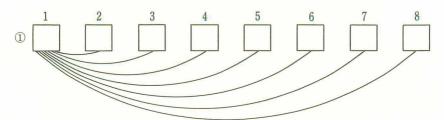
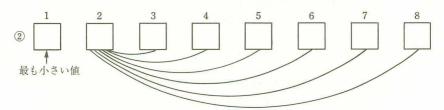
図2-1-3にその基本的なソートアルゴリズムを示します．

1と2～8を比較しながら1の方が大きければ値を入れかえる.
最終的に1には1～8の中で最も小さい値が残る.

2と3～8を比較しながら2の方が大きければ値を入れかえる.
最終的に2には2～8の中で最も小さい値が残る.

図2-1-3　バブルソートのアルゴリズム

1番から8番まで8個の数値があった場合，まず1番めと2～8番めの値とを順に比較していきます．その際，もし1番の値の方が小さければ1番とその数を入れ換えます．すべての数と比較し終ると，当然1番には最も小さい値が残っていることになります．

次に，2番めと3～8番の値を比較します．この時も同様の作業を行なうと，2番めには2～8番の数の中で最も小さい数が残ります．今度は3番と4～8番という具合にどんどん比較交換を繰り返していくと最終的には1番から8番までの小さい数字の順に並びます．

それでは，これを実際にプログラミングしてみましょう．数値を用意するのが面倒ですから，BDS Cで用意されている乱数発生用のrand関数を用い，ランダムな100個の数値を発生させて配列arrayに格納しておき，それをソートさせる事にしましょう．まず，リスト2-1-4に配列を用いてソートする例を示します．

リスト2-1-4　bsort1.c

```
#include <bdscio.h>

#define ARMAX   100
main()
{
        int     i;
        int     array[ARMAX];
        for (i=0; i<ARMAX; i++)
                array[i]=rand ();

        bsort1 (array,ARMAX);

        for (i=0; i<ARMAX; i++)
                printf ("%d¥t",array[i]);
}

bsort1(arr,n)
        int     arr[];
        int     n;
{
        int     i,j;
        int     tmp;

        for (j=0; j<n-1; j++)
        {
                for (i=j+1; i<n; i++)
                {
                        if (arr[j]>arr[i])
                        {
                                tmp = arr[i];        ) arr〔i〕とarr〔j〕
                                arr[i] = arr[j];     } の交換
                                arr[j] = tmp;        )
                        }
                }
        }
}
```

ここでソートの部分はbsort1という関数で，引数のarrはソートしたい整

数配列の先頭アドレス，nは配列要素の数です．なお，プログラムの先頭でARMAXという文字列に100を定義し，ソートする整数の数を100個にしています．ソート数を変える場合にはこの値を変更してください．

それでは次にこれをポインタを使って書き直した例をご紹介します．

リスト2-1-5 bsort2.c

```
#include <bdscio.h>

#define ARMAX   100
main()
{
        int     i;
        int     array[ARMAX];
        for (i=0; i<ARMAX; i++)
                array[i]=rand ();

        bsort2 (array,ARMAX);

        for (i=0; i<ARMAX; i++)
                printf ("%d¥t",array[i]);
}

bsort2(pnt,n)
        int     *pnt;
        int     n;
{
        int     *spnt;
        int     i,j;
        int     tmp;

        for (j=0; j<n-1; j++)
        {
                spnt=pnt+1;  ← spntにはすぐとなりの要素のアドレスをセット．
                for (i=j+1; i<n; i++)
                {
                        if (*pnt>*spnt)
                        {
```

```
                                        tmp = *spnt;     )
                                        *spnt = *pnt;    } *spntと*pntの交換.
                                        *pnt = tmp;      )
                                }
                                spnt++;
                        }
                        pnt++;
                }
}
```

関数bsort2が変更した部分で，その他の部分は全く同じです．ポインタを使うとどうしてもリストがわかりにくくなりますが，やむを得ないでしょう．

さて，関数bsort 2では比較変更する整数を指すポインタとして pnt と spntを使っています．pnt は引数ですから，初期値として配列の先頭アドレスがあらかじめ設定されていますが，spntはローカル変数ですから必ず値をセットする必要があります．pnt は基準位置，spnt は比較位置を指すポインタとして使っています．

```
pnt     ←0番めの整数のアドレス
spnt    ←1番めの整数のアドレス
```

からスタートします．BDS C，α-C では int は2バイトですから，spnt＝pnt＋1 ではなく，spnt＝pnt＋2 にしなければならないのではないかと思われる方もいると思います．確かにアセンブラでプログラムを作る場合は当然そうしなければ大きなバグを残してしまいますが，C言語のポインタの加減算では，対象とする変数の1要素分が1になる事になっており，これが正しいのです．

16ビット，32ビットCPUを持つコンピュータでは int が必ずしも2バイトではなく，他のビット数を持つことも考えられるわけですから，移植性を考えると当然とも言えるわけです．

リスト2-1-4と2-1-5は完全に対応していますから，是非両者を見比べ，ポインタの感覚をつかんで頂きたいと思います．実行の結果を実行例2

-1-6に示します（両者の実行結果は全く同一です）．

なお，気になるオブジェクトサイズですが，bsort1，bsort2の関数部分はそれぞれ226バイト，204バイトとなりました．100個程度のデータでは計測できませんが，実行時間にも差がでているでしょう．

なお，蛇足ですが，BDS Cにはqsortというソート用の関数があり，大量のデータを高速に並べ変える必要がある場合にはこちらを用いた方が良いでしょう．qsortはシェルソートアルゴリズムを用いています．

実行例2-1-6　バブルソートの実行結果

```
A>BSORT1

0      0      12     16     25     32     32     34     44     50
64     64     68     72     89     100    100    104    128    136
144    145    178    200    200    256    384    400    768    768
800    1650   1792   1792   1856   2262   2292   3301   3584   3840
3968   4257   4269   4293   4329   4414   5123   6602   6671   7168
7936   7936   8259   8282   8331   8402   8476   8757   8829   9231
10247  11769  12948  12984  13542  13599  14592  15872  15872  16406
16518  16548  16564  17209  17515  17658  18463  18512  18658  18945
19719  20494  20890  20999  22140  23283  25640  25713  25856  26115
26829  26883  27454  29184  29441  29697  30976  31232  31744  32000

A>
```

2-2　コマンドラインの受け取り方

C言語でプログラムを作成する際，その多くの場合にコマンドラインからの文字列を利用します．C言語ではこの受け取り方はほぼ固定されており，OSを問わず，移植性が高い部分です．

このサンプルプログラムはコマンドラインからの文字列を受け取り，そのまま画面に表示するものです．プログラム作成中に，特別な文字や記号などを引数としたい場合に，それが有効かどうか判らないことがあります．そのような時，リストのような簡単なプログラムは意外に役立ちます．

リスト2-2-1　args1.cと実行例

```
#include        <bdscio.h>

main(argc,argv)
        int     argc;
        char    *argv[];
{
        int     i;
        printf ("argc = %d¥n",argc);
        for (i=1; i<argc; i++)
        {
                printf ("arg%d=¥"%s¥"¥n",i,argv[i]);
        }
}
```

〈実行例〉

```
A>args1 MITARAI 31 ABCD 1 2 3

argc = 7
arg1="MITARAI"
arg2="31"
arg3="ABCD"
arg4="1"
arg5="2"
arg6="3"

A>
```

さて，mainの関数宣言の部分に注目してみましょう．

```
main(argc,argv)
  int   argc;
  char  *argv[];
```

というのはコマンドラインからの引数（文字列）を受け取る際の定石です．main関数は実行が開始される最初ですから，mainを呼び出す上位の関数とい

うものは存在しないわけですが，C言語ではmain関数であろうとも他の関数と全く区別しませんので，システムが自動的にコマンドライン文字列を解析し，あたかもmainに対する上位関数があるかのようにその結果をmain 関数に渡すわけです．

ここで，argc，argvは

argc	引数の数
argv	引数文字列のポインタ変数配列の先頭アドレス

となります．要は，

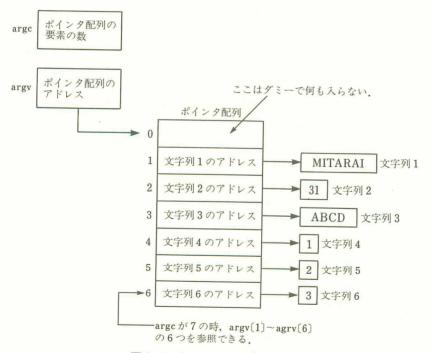



図2-2-2　argvと文字列の関係

という構造になっており，2段階のメモリ参照で，文字列そのものを参照することができるわけです．実際のプログラムでは

argv[1] ―1つめの文字列の先頭アドレス
argv[2] ―2つめの文字列の先頭アドレス
⋮
argv[n] ―n番めの文字列の先頭アドレス

となります．文字列そのものを表示したければ，

```
printf (argv[n]);
```

とするだけで，n番めの文字列を画面に出力できます．

この考え方は初心者の方にはなかなか難しいのですが，定石ですので，判らなければそのまま覚えてください．そのうちにメカニズムがはっきり判るようになると思います．なお，printf関数は一般に，その第一引数は文字列をじかに“ ”で囲んで引数とする場合が多いのですが，そのような記述も実はこの文字列そのものがプログラム中のどこかに埋め込まれ，printf関数への実際の引数は文字列のアドレスとなっています．つまり，文字列を記述するという事は文字列への先頭アドレスを書くのと同義なのです．

なお，リスト2-2-1では

```
printf("arg%d=¥"%s¥"¥n",i,argv[i]);
```

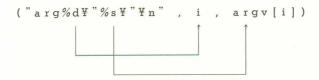
となっています．ここではprintfの重要な機能のいくつかを用いています．まず，( )の中を分解してみると，

(a) "arg%d=¥"%s¥"¥n"
(b) i
(c) argv[i]

の3つに分けられることが判ります．ここで(a)は文字列ですが，中にいくつか特殊な記号が入っています．すでに出てきたものもありますが，もう一度確認しておきましょう．

%d　　%は制御文字で，%の直後にある文字によりさまざまな機能を持たせることができます．dは10進整数の指定で，文字列の後にある引数（この場合はｉ）の値を10進整数に変換して表示します．

%s　　%の後がｓの場合，引数の値を文字列へのポインタ（文字列の先頭アドレス）と解釈し，その文字列を表示します．この場合はargv[i]，すなわちｉ番めのコマンドラインの文字列となります．

これらの%は文字列の中にそのまま組み込まれて，printf文実行の際に変換されて表示されます．この時，制御文字列と引数の関係は，

となり，制御文字がでてくるたびに次々と新たな引数を参照していくことになります．このようにprintfは引数の数が一定しない特殊な関数で，文字列の内容と引数の数を対応させなければ正しく動作しません．

なお，文字列中にでてくる¥は次のような意味を持っています．

¥"　　文字列はその最初と最後を"で囲みますから，"自身を文字列中に含ませることができません．そこで文字列では¥記号を特殊記号として扱い，¥"は文字"そのものとして扱います．

¥n　　改行を示します．

これらの¥を伴うものは，¥の文字そのものが残るわけではなく，コンパイルされた時点で変換されて目的のコードに変わります．つまり，これはprintf文の機能ではなく，コンパイラの機能になるわけです．

さて，このプログラムは簡単ですが，基本的なC言語の機能をたくさん含んでおり，非常に良いサンプルだと思います．特に初心者の方は判るまでテストしてみることをおすすめします．

さて，main関数の引数として一般にはリスト2-2-1のように宣言する場合が多いのですが，argvをポインタ変数として宣言することがあります．機能としては全く同じなのですが，考え方が異なります．

```
(1)  main(argc,argv)
        int  argc;
        char *argv[];

(2)  main(argc,argv)
        int  argc;
        char **argv;
```

(1)は文字列へのポインタ変数配列の先頭アドレスがargv（の内容）であるという解釈をする場合，(2)はargvというポインタ変数の初期値がポインタ変数配列の先頭に設定されていると解釈する場合です．

リスト2-2-3に(2)の方法を用いてリスト2-2-1を書き直したものを示しますが，少しプログラムが変わっているのに気付かれたと思います．ポインタ配列へのポインタとして宣言した場合，argvはポインタ変数として扱う方が自然になります．

なお，＊＋＋argvはargvの内容をインクリメントした後，その内容をアドレスとした内容になります．

リスト2-2-3 args2.c

```
#include        <bdscio.h>

main(argc,argv)
        int     argc;
        char    **argv;
{
        int     i;
        printf ("argc = %d¥n",argc);
        for (i=1; i<argc; i++)
        {
                printf ("arg%d=¥"%s¥"¥n",i,*++argv);
        }
}
```

コマンド引数の受取りは非常に重要な部分で，慣れれば簡単なのですが，ポインタ配列というやや高度な概念を含むため，初心者の方は意外に悩むことが多いものです．本項でしっかり把握してください．

2-3 ファイルの入出力

さて，ファイルの入出力はいくつもの標準関数に支えられて，大変便利になっています．BASICなどに較べるとはるかに機能も豊富ですが，そのぶん関数の種類が多く，どれを使えば良いか悩むこともしばしばです．

1つ1つの関数について，マニュアルを良く読むことも必要ですが，まず基本的ないくつかの関数をしっかり理解することも重要です．

〈2種類のファイルIO〉

一般的なC言語には低レベルファイルIO（Input Output＝入出力）とバッファードファイルIOという2つのファイル入出力用の関数群が用意されています．これらは基本的には次のように考えれば良いでしょう．

低レベルIO　　— ファイルの物理的な記録単位（CP/Mでは128バイト）での入出力を行なう場合

バッファードIO　— ファイルと1文字単位でデータの出し入れを行なう場合

多くの場合，バッファードファイルIOの方が操作は簡単ですが，バッファードと言えどもその中で低レベルファイルIOを利用しているのですから，処理そのものの速度はどうしても遅くなります．ファイルコピーのようにどちらを利用してもできるようなプログラムの場合には当然低レベルIOの方が処理速度は速くなります．

〈低レベルファイルIO〉

ファイルをブロック単位で扱う場合に使いますので，ファイルのランダムアクセス，ブロック転送などに有効です．

低レベルファイルIOとして重要なのは次の4つの関数です．

open　— 既にディスク中に存在するファイルを読み書きするためにファイルをオープンします．

creat — ファイルがある場合には消去し，書き込みのために新しくファイルをオープンします．

close — ファイルをクローズします．

read　— ファイルをブロック単位で読み出します．

write — ファイルにブロック単位で書き込みます．

seek　— ランダムアクセスなどのため，ファイルの読み書きポインタを変更します．

まず，openが既に存在するファイルをオープンし，読み出し，書き込み，あるいはその双方に備えるのに対し，creatはファイルがある場合には消去してから書き込み用にオープンします．従って，creatでは同じ名前のファイルは強制的に消去されますから，注意が必要です．open，creatによってオープンされ，書き込み処理が終了したファイルは必ず最後にcloseによってクロ

ーズしなければなりません．

read, write はそれぞれファイルからの読み込み，ファイルへの書き込みを行ないます．BDS C ではこの単位は128バイトになっており，1バイトだけが必要な場合にも128バイトいっぺんに読み書きする必要があります．

一応，これらの関数の存在と簡単な機能だけを知っておけば，後はプログラム作成時にマニュアル通りに記述していくだけでも何とかなります．

まず，簡単な例として，ファイルの内容を16進数とアスキーの両方で表示するプログラムを作成してみます．

リスト2-3-1　cdump.c ファイルダンププログラム

```
 1:  #include        <bdscio.h>
 2:
 3:  #define         ERROR   -1
 4:  #define         INPUT   0
 5:
 6:  main(argc,argv)
 7:          int     argc;
 8:          char    *argv[];
 9:  {
10:          char    buf[128];       /* sector buffer */
11:          int     fd;
12:          int     sz;             /* size of sectors */
13:          unsigned adr;
14:
15:          if (argc!=2)
16:          {
17:                  printf ("Usage: cdump filename");
18:                  exit ();
19:          }
20:
21:          fd = open (argv[1],INPUT);
22:          if (fd == ERROR)
23:          {
24:                  printf ("Can't open %s",argv[1]);
25:                  exit ();
26:          }
27:
28:          adr = 0;
29:          while (1)
30:          {
31:                  sz = read (fd,buf,1);
32:                  if (sz < 1)     break;  /* file end */
```

```
33:                     dump (buf,adr);
34:                     adr += 128;
35:             }
36:             close (fd);
37:     }
38:
39:
40:     dump(buf,adr)
41:             char    *buf;
42:             unsigned adr;
43:     {
44:             int     i,j;
45:             for (i=0; i<128; i+=16)
46:             {
47:                     printf ("%04x  ",adr+i);        /* address output */
48:
49:                     for (j=0; j<16; j++)            /* hex output */
50:                             printf ("%02x ",buf[i+j]);
51:
52:                     printf ("  ");
53:
54:                     for (j=0; j<16; j++)            /* ascii output */
55:                     {
56:                             if (buf[i+j]<' ')       printf (".");
57:                             else                    printf ("%c",buf[i+j]);
58:                     }
59:                     printf ("¥n");
60:             }
61:     }
62:
```

［15行～19行］

プログラムの実行方法が分らなくなったり，誤った指定をした場合に，CDUMPの使用方法を示す部分です．15行めで（argc！＝2）となっています．argcが2以外の場合，つまりファイル名が指定されなかったり，あるいは2つ以上の文字列をプログラム名に続けて指定してしまった場合はすべてこのUsage：を表示して終りになります．！＝は一致しない時に真となる比較演算子です．

exit（　）はプログラムを中断し，強制的に終了する場合に使います．

［21行～26行］

関数openを実行し，ファイルが存在しないなどの理由でエラーとなった場合にはそのメッセージを表示して終ります．

［28行］

ダンプアドレスを初期化します．これはダンプの際，左端に表示するアドレスが0からスタートするように指定します．

［29行〜35行］

ここがプログラムの本体部分です．31行でファイルをリードし，もし実際に読み込んだセクタ数（ブロック数）が指定した読み込みセクタ数の1より小さかった場合，ファイルの終了と考えられます．その場合にはこのループから抜け出します．

もし，正しく読み込めた場合にはその内容を関数dumpに送ってダンプし，ダンプアドレスadr を128カウントアップしておきます．

［36行］

終了ですので，ファイルをクローズします．

実行例2-3-2　cdump.cの実行例

```
A>cdump

Usage: cdump filename

A>cdump cdump.c

0000  23 69 6E 63 6C 75 64 65 09 3C 62 64 73 63 69 6F  #include.<bdscio
0010  2E 68 3E 0D 0A 0D 0A 23 64 65 66 69 6E 65 09 09  .h>....#define..
0020  45 52 52 4F 52 09 2D 31 0D 0A 23 64 65 66 69 6E  ERROR.-1..#defin
0030  65 09 09 49 4E 50 55 54 09 30 0D 0A 0D 0A 6D 61  e..INPUT.0....ma
0040  69 6E 28 61 72 67 63 2C 61 72 67 76 29 0D 0A 09  in(argc,argv)...
0050  69 6E 74 09 61 72 67 63 3B 0D 0A 09 63 68 61 72  int.argc;...char
0060  09 2A 61 72 67 76 5B 5D 3B 0D 0A 7B 0D 0A 09 63  .*argv[];..{...c
                          （後略）
A>
```

〈バッファードファイルIO〉

さて，低レベルファイルIO関数がディスクファイルの内容をブロック単位

でしか扱えないのに対し，バッファードファイルIOでは1文字単位でファイルを扱うことができます．しかしながら，CP/Mではあくまでディスクとのやりとりは128バイトという固定されたサイズでしかできないのだという事は知っておく必要があります．バッファード(Buffered)という意味もまさにそこにあり，1文字入力の場合であればディスクからブロック単位のデータを読み取って一旦バッファー(メモリ)に蓄え，そこから1文字1文字切り取ってくることを示しています．

高級なファイルIOですから，使い方はかえってやさしいのですが，まずどのような関数があるか説明します．

fopen	—	既に作成されているファイルをオープンします．
fcreat	—	同一名のファイルがある場合にはそのファイルを消去し，書き込むためにファイルをオープンします．
fclose	—	ファイルをクローズします．
getc	—	ファイルから1文字読み込みます．
putc	—	ファイルに1文字書き込みます．
fgets	—	ファイルから1行読み込みます．
fputs	—	ファイルに1行書き込みます．
fprintf	—	ファイルへの出力に対し，printfと同じ機能を持つ関数です．
fscanf	—	ファイルからの入力に対し，scanfと同じ機能を持つ関数です．

この他にもいくつか関数がサポートされていますが，重要なものは上記の9つで，これらをうまく使うことでほとんどの処理が可能です．

さて，バッファードファイルIOの考え方で一番重要なのは，文字が中心に考えられている事です．つまり，上記の関数はすべて，英数字などを格納したアスキーファイルを扱う事を前提に考えられており，COMファイルなどのバイナリファイルを扱う際には必ずしも便利ではありません．

例えば，アスキーファイルではそのファイルの終端文字として，コントロールZ(1AH)が用いられますが，この文字が現れるとファイルの終了とみな

す関数（fgets）とそのまま1AHとして返す関数（getcなど）がありますから，使用する際には注意が必要です．

さて，このバッファードIO関数を使う場合，必ずbdscio.hという標準ヘッダーファイルを先頭でインクルードしなければなりません．bdscio.hの中ではこれらの関数で読み書きのために使うバッファーを確保する構造体を宣言していますので，これにより，バッファードIO関数を利用する時には，

```
FILE iobuf;
```

などと宣言すれば自動的にバッファードファイルIO用のバッファー，及びその状態などを記憶しておくフラグ・カウンタなどのメモリが確保されます．何が何バイトで，といった事を考える必要はありません．

リスト2-3-3はディスクからファイルを読み，それを行ごとにナンバーを付けて別ファイルとして格納するプログラムです．コマンドラインから，

```
A>fnum file1 file2
```

とすると，file1の内容に行番号を付加し，file2として出力します．

入力のためにファイルからの行入力関数fgets，出力に書式つきファイル出力関数fprintfを利用している点に注意してください．

リスト2-3-3　fnum.c 行番号付加プログラム

```
 1:  #include        <bdscio.h>
 2:
 3:  FILE    fpin;
 4:  FILE    fpout;
 5:
 6:  main(argc,argv)
 7:          int     argc;
 8:          char    *argv[];
 9:  {
10:          char    buf[200];
11:          int     lcnt;
12:
```

```
13:         if (argc != 3)  printerr ("Usage: fnum infile outfile");
14:
15:         if (ERROR == fopen (argv[1],fpin))      printerr ("Can't open.");
16:         if (ERROR == fcreat(argv[2],fpout))     printerr ("Can't create.");
17:
18:         lcnt = 1;
19:         while (fgets (buf,fpin))
20:         {
21:                 if (ERROR == fprintf (fpout,"%5d:¥t%s",lcnt,buf))
22:                         printerr ("Write error.");
23:                 lcnt++;
24:         }
25:         fclose (fpin);
26:         fclose (fpout);
27: }
28:
29: printerr(msg)
30:         char    *msg;
31: {
32:         printf (msg);
33:         exit ();
34: }
35:
```

［3，4行］

ここでバッファードIO関数を用いるための入出力用ファイルを宣言しておきます．ここでは入力用としてfpin，出力用としてfpoutの2つを宣言していますが，このように入出力のどちらにも同じ書式です．なお，この宣言は関数の外側で行なわれていますので，どの関数からも自由にアクセスできる外部変数になります．関数の内部で宣言すればローカル変数として，その関数の内部でだけ利用可能となります．

［10，11行］

10行めでは1行を格納するための行バッファーを200バイト宣言しています．fgetsでは1行の長さをチェックしませんので，もしこれより長い行がファイル中に存在すると暴走する可能性があります．一般にはこれより長いものはめったにありませんが，信頼性が特に要求されるような用途にはもっと確実な方法を選ぶ必要があります．

11行で宣言しているlcntは行カウンタです．1行読むごとにインクリメントされます．

[13行]

13行めで行なっているのはプログラム起動時のコマンドラインからの引数チェックです．fnum.cは必ず2つのファイル名を指定する必要がありますから，もしそれ以外（すなわちargcが3以外）であったら，Usage：以下を表示して終了します．printerrはそのためのメッセージを表示してプログラムを終了する関数です．

[15，16行]

15，16行ではファイルの入出力のため，2つのファイルをオープンします．fpinを入力用に，fpoutを出力用にしています．ここで，ERRORというのは定数で，－1を示しています．一般にこのような定数は#define文で宣言しなければなりませんが，非常に一般的な定数はbdscio.hの中で宣言されているので，宣言しなくても利用できます．このような定数には他に，

```
NULL     0
EOF      -1
OK       0
TRUE     1
FALSE    0
CPMEOF   0x1a（コントロールZのアスキーコード）
```

などがあります．

[18行]

行カウンタlcntの値を1に初期化します．

[19行]

fgets関数によりfpinで指定されているファイルから1行を読み込み，bufへ格納します．かつもしその戻り値が0以外の場合には{}の中を繰り返します．fgetsはCR，LF（改行コード）があるとそれをLF(¥n)だけに置き直

し，かつ最後にC言語での文字列終端文字として0を追加します．

なお，ファイルが終了した場合，fgetsは0を返します．whileの条件は0以外（真）の場合ループし，0（偽）の場合，ループを終了しますので，このような場合には都合が良くなります．

［21，22行］

プログラムの本体部分です．fgetsで読み込んだ1行に行番号を付加して，fpoutのファイルに出力します．fprintfは書き込みエラー時に－1を返しますから，ここでERRORと比較し，エラーなら強制終了します．

fprintfなどは良くERRORチェックなしでも利用されることがあり，そのような場合にはファイルの書込みチェックはされません．これは書き込む位置が1箇所でなく複数の箇所になる場合など，プログラムサイズが大きくなり，動作速度も遅くなりますので，意識して省かれるわけです．しかし，書き込みエラーはしばしば発生するトラブルであり，やはりエラーチェックはすべきでしょう．

［23行］

ラインカウンタをインクリメントします．

［25，26行］

プログラムの終了処理です．入力用ファイルfpin，および出力用ファイルfpoutをクローズします．

［29～34行］

文字列を表示してプログラムを強制終了する関数printerrです．

実行例 2-3-4　fnum.cの実行例

```
A>fnum

Usage: fnum infile outfile

A>fnum fnum.c fnum.n

A>type fnum.n

    1:  #include        <bdscio.h>
    2:
    3:  FILE    fpin;
    4:  FILE    fpout;
    5:
    6:  main(argc,argv)
           (後略)
A>
```

2-4　文字列の処理

BDS Cに限らず，C言語では文字列の処理がBASICなどと異なるので，最初のうちはとまどう事もしばしばです．BASICでは文字変数は文字列への先頭アドレスと文字数という2つの要素で記憶されている場合が多いのですが，Cではあくまで文字列は文字型（char）の配列としてのみ扱われ，文字列の最後に0(16進のゼロ)を置き，この終端文字と先頭アドレスで扱います．

この方法は文字列をアドレス(2バイト)だけで受け渡しできる点で有利ですが，文字列操作では必ずしも便利ではなく，文字列の先頭何文字かを取り出したりする場合には文字を1字ずつ追いかけなければならないため実行が遅くなるという欠点も持っています．また，文字列の中に0（16進のゼロ）を使うことはできませんから，プログラムオブジェクトなどのバイナリデータを扱う場合には特に不便です．一般にC言語でバイナリデータを扱うプログラムを作成する場合，自分でいくつかの文字列操作用関数を作る必要があ

ると考えてよいでしょう．本項での文字列操作とは一般の文字や文章の操作であると考えてください．

〈文字列処理の基本ファンクション〉

文字列をそのまま別の場所に移したり，付け足したり，比較したりする場合は非常に多いものです．これらには次のような関数を利用します．

(1)　strcpy (string copy)

文字通り，文字列をコピーする場合に用います．

```
strcpy (t,s)
```

とした場合，sの文字列をtにコピーします．この場合，tもsも文字列へのアドレスで，tはsをコピーできるだけの十分な長さを持っている必要があります．

(2)　strcat (string concatenate(文字列連鎖))

文字列を連結する場合に使います．

```
strcat (s,t)
```

とした場合，sの文字列の後にtを付加します．sはtが後に付加されるだけの十分な長さを持っている必要があります．

リスト2-4-1　string.cと実行例

```
#include <bdscio.h>

main(argc,argv)
        int     argc;
        char    *argv[];
{
        char    buf[100];
```

```
        strcpy (buf,argv[1]);
        strcat (buf,argv[2]);
        printf (buf);
}
```

〈実行例〉

```
A>string ABC DEF
ABCDEF

A>string JKLMN OPQRST
JKLMNOPQRST

A>
```

リスト2-4-1は，コマンドラインの入力した2つの文字列を合成して表示します．2つ入力がない場合のチェックは行なっておりませんので必ず，string XXXX YYYY というフォーマットで実行してください．

(3) strcmp (string compare)

文字列同士を比較し，同一かどうかをチェックします．

```
strcmp (s1,s2)
```

とした場合，一致すれば0，一致せず最初に異なった文字について，s1>s2なら正の値，s1<s2なら負の値を返します．

リスト2-4-2　strcmp.c と実行例

```
#include        <bdscio.h>

main(argc,argv)
        int     argc;
        char    *argv[];
{
        if (strcmp (argv[1],argv[2]))
```

```
                printf ("not same");
        else
                printf ("same");
}
```

〈実行例〉

```
A>strcmp ABCDEFG ABCDEFG
same

A>strcmp ABCDEFG ABCDEF
not same

A>
```

リスト2-4-2は，コマンドラインから入力した2つの文字列が等しいかどうかをチェックし，等しければ“same”，同じでなければ“not same”と表示します．

(4) strlen (string length)

文字列の長さを求めます．終端文字の0は含みません．

この関数はポインタの項で解説したstrlen2と全く同じものですので，リスト2-1-1，2-1-2を参照してください．

これらの関数は簡単なもので，自作してもほとんど数行でできるようなものばかりですが，非常に重要です．これらを使ってみるのが，文字列処理をマスターする第1歩でしょう．これら以外にもいくつかの関数がありますが，その使用頻度は上記の4関数程高くはありません．

〈ユーティリティ関数〉

文字列処理関数は非常に便利なものですが，BASICに慣れたかたなどは，かなり物足りなく感じる場合も多いものです．そこで，標準関数にないBASICライクな関数をいくつか紹介しておきます．文字列というものがどう扱われ，

どのようにすれば文字列処理が可能なのか良いサンプルになると思います．特にここで紹介する関数が便利なのは，BASICのプログラムをCに移植する場合です．

```
char *left(str,len)
  char *str;
  int  len;
```

BASICのLEFT$関数に対応する関数です．strに文字列へのアドレスをセットし，lenに文字列の先頭からの長さを指定すると，文字列strがその文字数までで打ち切られ，strのアドレスが戻り値として返ります．元の文字列は失われることになりますから注意してください．

なお，指定の長さより与えられた文字列の方が短ければそのまま返ります．

リスト2-4-3　left.cと実行例

```
#include        <bdscio.h>

main(argc,argv)
        int     argc;
        char    *argv[];
{
        printf (left (argv[1],3));
}
```

左から3文字を切り取る．

```
left(str,len)
        char    *str;
        int     len;
{
        char    *s;
        s = str;
        while (len--)
        {
                if (!*s++) return (str);
```

```
	}
	*s = '¥0';			/* string end code */
	return (str);
}
```

〈実行例〉

```
A>left ABCDEFG
ABC

A>left JKLMNOP
JKL

A>
```

サンプルプログラムでは，コマンドラインから入力した文字列を先頭3文字で切り取り，表示します．

```
char *right(str,len)
  char *str;
  int  len;
```

BASICのRIGHT$関数に対応する関数です．strに文字列へのアドレスをセットし，lenに文字列最後からの文字数をセットするとその位置のアドレスを戻り値として返します．文字列の途中から使うことになりますので，元の文字列は失われません．

指定の長さより文字列が短かければ文字列はそのまま戻ります．

リスト2-4-4 right.cと実行例

```
#include	<bdscio.h>

main(argc,argv)
	int	argc;
	char	*argv[];
```

```
{
        printf (right (argv[1],3));
}
```

右から3文字切り取る.

```
right(str,len)
        char    *str;
        int     len;
{
        int     i;
        char    *s;
        s = str;
        i = 0;
        while (*s++)    i++;
        if (i<len)      return(str);

        while (len--)   s--;
        return (--s);
}
```

〈実行例〉

```
A>right ABCDEFG
EFG

A>right JKLMNOP
NOP

A>
```

リスト2-4-4はコマンドライン引数の文字列の右3文字を切り取ります.

```
char *mid(str,pos,len)
  char *str;
  int  pos;
  int  len;
```

BASICのMID$関数に対応する関数です．strに文字列へのアドレスをセットし，posに切り取りたい文字の先頭からの位置，lenに長さを指定すると，目的の文字列へのアドレスが戻り値として返ります．元の文字列は失われます．

なお，指定の位置より文字列が短ければヌルストリング（終端文字だけで内容のない文字列）が返り，指定の長さより文字列が短かければ，途中で打ち切られた文字列となります．

リスト2-4-5　mid.cと実行例

```
#include        <bdscio.h>

main(argc,argv)
        int     argc;
        char    *argv[];
{
        printf (mid (argv[1],2,3));
}
                    ← 2文字めから3文字切り取る.

mid(str,pos,len)
        char    *str;
        int     pos,len;
{
        int     i;
        char    *s;
        i = 0;
        while (pos--)
                if (!*str++)    return (--str);

        s = --str;
        while (len--)
                if (!*s++)      return (str);
        *s = '¥0';
        return (str);
}
```

〈実行例〉

```
A>mid ABCDEFG
BCD

A>mid JKLMNOP
KLM

A>
```

リスト2-4-5はコマンドライン文字列の2文字めから3文字分を切り取って表示します．

2-5 構造体の使い方

構造体もC言語の特徴の1つです．BASICなどには無い概念なので，最初はとまどいますが，決して難しいものではなく，むしろプログラムを読みやすく，判りやすくしてくれます．

ここではサンプルとして，役には立ちませんが簡単な電話帳プログラムを作成してみましょう．名前・年齢・電話番号の3つがデータで，2人分のデータが入力されるとその内容を表示して終了します．

このプログラムは構造体を使わなくても実現できますので，比較の意味でまず配列で作成したものを説明します．

リスト2-5-1 memlist1.cと実行例

```
#include <bdscio.h>

#define MEMBERS 2

main()
{
        char    name[MEMBERS][25];  )
        int     age[MEMBERS];       } 別々の配列.
        char    tel[MEMBERS][15];   )
        int     i;
```

```
        for (i=0; i<MEMBERS; i++)
        {
                printf ("name->");
                gets (name[i]);
                printf ("age ->");
                scanf ("%d",&age[i]);
                printf ("tel ->");
                gets (tel[i]);
        }
        for (i=0; i<MEMBERS; i++)
        {
                printf ("%s (%d)  tel:%s¥n",name[i],age[i],tel[i]);
        }
}
```

〈実行例〉

```
A>MEMLIST1

name->T.MITARAI
age ->31
tel ->045-911-7427
name->R.MITARAI
age ->26
tel ->045-911-7427

T.MITARAI (31)  tel:045-911-7427
R.MITARAI (26)  tel:045-911-7427

A>
```

判りにくい部分は全くないと思います．名前を入れる配列name，年齢を入れる配列age，電話番号を入れる配列telの3つを用意します．勿論name，telは文字列ですから文字列の配列で2次元配列となります．

これらの配列に対し，文字列の行入力関数getsと入力関数のscanfを用いて値をキーボードから入力し，2人分入れるとその内容を画面に出力します．人数はMEMBERSで，2が指定されていますからこの値を変更すれば何人にでも対応させることができます．

このようなプログラムはBASICなどでは常套手段であり，配列要素の順

に，関連のあるデータを格納する方法は良く用いられます．しかし，ここでname，age，telというのは相互に関係の深い配列であるにも拘わらず，完全に独立しているため，その関係が明確ではありません．そこでC言語ではこの3つの配列をひとまとめにし，name[ ]，age，tel[ ]という型の異なる複数の要素を一つの要素として扱えるようになっています．これを構造体といいます．

構造体を用いて書き直したものをリスト2-5-2に示します．

リスト2-5-2　memlist2.c

```
#include <bdscio.h>

#define MEMBERS 2

struct  member
        {
                char    name[25];
                char    age;
                char    tel[15];
        };

main()
{
        struct member   list[MEMBERS];
        int     i;

        for (i=0; i<MEMBERS; i++)
        {
                printf ("name->");
                gets (list[i].name);
                printf ("age ->");
                scanf ("%d",&list[i].age);
                printf ("tel ->");
                gets (list[i].tel);
        }
        for (i=0; i<MEMBERS; i++)
        {
                printf ("%s (%d)  tel:%s¥n",list[i].name,list[i].age,list[i].tel);
        }
}
```

structからmain( )の前までが，構造体タグmemberの定義であり，以降

```
struct member list;
```

とするだけで，member 型の要素を持った構造体の宣言ができます．例のようにさらにそれを配列にしておけば，構造がはるかに明確に指定できます．

プログラムそのものは長い文字列を書くことになりますので，ちょっと見ずらく感じられますが，リストが長くなってくるとはるかに判りやすくなります．

このように複数の配列を対応させて扱うような用途には，構造体は非常に便利なものです．

なお，構造体の要素への読み書きは

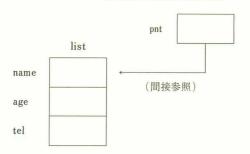
```
list.name     ：構造体変数 list の要素 name
list.age      ：構造体変数 list の要素 age
list.tel      ：構造体変数 list の要素 tel
```

のように，変数名と要素名の間をピリオドでつなぎます．ポインタにより間接的にアクセスする場合には，

```
struct member *pnt;
pnt=list;
```

などとしておいて，

```
pnt->name
pnt->age
pnt->tel
```

のようにーと＞の2つの文字で結合させます．このポインタによる間接アクセスは混乱を招きやすいのですが，次のように考えれば良いでしょう．

図のようにpntという変数の内容（アドレス）が指しているのがmember型の(listという名前の）構造体です．listというのは名前であると同時にこの構造体の先頭アドレスを示す定数です．つまり，直接に定数から読み書きする場合には，

```
list.name
```

とすれば良く，間接的に（ポインタ）変数の内容から参照する場合には

```
pnt->name
```

となるわけです．ところが文字列の受け渡しなどでは，19ページのリスト2-1-1などのように引数が配列（の先頭アドレス）であることを宣言すると，その引数については間接的な読み書きにもかかわらず

```
s[i]
```

などのようにあたかも直接的な読み書きであるかのように書くことができますが，構造体ではこのような記述ができません．初心者が誤りやすいポイントですので，構造体の場合は直接参照か間接参照かを意識し，プログラムを書き分けるようにしてください．

配列に比べると，構造体は特殊な操作をするような印象を与えますが，よく考えてみると関係のあるものを集めてブロック単位で扱う方がはるかに自然で合理的であることが納得できると思います．

第 3 章

BDS C特有の問題点

本章はBDS C，α-Cにおいて，問題になりやすい部分について解説します．その多くはこれらのコンパイラがC言語のサブセットであることから起るものです．

ただ，これらの欠点があるからBDS C，α-Cは良くないという結論は短絡的で，どのようなコンパイラでも固有の問題というものはあるものです．これらの欠点のため，コンパイルが速い，バグが少ないという，大きなメリットを忘れてはならないと思います．

なお，本章のような内容を読まれて，「このような機能削除版のコンパイラで果して正しいC言語の学習ができるのだろうか」と思われるかも知れません．あらかじめ断言しておきますが，BDS C，α-Cで覚え，経験したC言語のノウハウが他のCコンパイラで役に立たないということは決してありません．ご安心ください．

3-1 コンパイルエラー

BASICは言語仕様上エラーの検出がたやすく，有名なSyntax error から始まり，多くの誤りを完全に指摘してくれます．ところが，C言語はフリーフォーマットであるため，行というプログラム分割単位を持たず，コンパイル時に起るエラーがどこでどのようにして発生したか判定しにくいという欠点があります．

エラーメッセージとして最も頻出するのは文末にセミコロン（；）をつけ忘れる

Missing semicolon

だと思います．これは既に読者の方々も経験ずみなのではないでしょうか．

また，エラーメッセージが予想以上に多く発生することもしばしばあります．これは大抵の場合，はじめにあるエラーが連鎖的に後のエラーを招いたもので，多くは最初のエラーを取り除くと消えます．

リスト3-1-1 大量のエラーが発生する例

```
 1: #include        <bdscio.h>
 2:
 3: main()
 4: ←――――――ここにカーリーブレイス｛が落ちている．
 5:             int     i,s;
 6:             char    f;
 7:
 8:             for (s=2; s<=100; s++)
 9:             {
10:                     f=0;
11:                     for (i=2; i<=s-1; i++)
12:                     {
13:                             if (s%i==0)
14:                             {
```

```
15:                                        f=1;
16:                                        break;
17:                              }
18:                    }
19:                    if (f==0)            printf ("%d¥t",s);
20:          }
21: }
```

〈コンパイル例〉

```
A>cc f:error1
BD Software C Compiler v1.50a (part I)
   5: Missing from formal parameter list: i
   5: Missing from formal parameter list: s
   6: Missing from formal parameter list: f
  21: Unmatched right brace

A>
```

カーリーブレイス｛がないため変数の宣言がすべてエラーになる．

｛｝が対応せず｝が1つ多いため21行めでエラーとなる．これと5，6行めのエラーを考慮すると4行めのミスは容易に判明する．

エラーの対処方法は場合，場合によって異なり，その都度エラーメッセージを解析していくことになりますが，基本的には次のような誤りが多いと思われます．

(1) ｛と｝のアンバランス

C言語では｛が出現するたびに字下げを行ない，｝により，もとの桁位置に戻すようにすると，｛｝のバランスが崩れるのを防ぐことができます．これについては本書のリストのほとんどが実施していますので，参考にしてください．

(2) （と）のアンバランス

これはBASICでも同じです．複雑な式などではどうしても誤りやすいので数えてみた方が良いでしょう．なお，

```
if ((kansuu1 () && flag1)||(kansuu2 () && flage2))
```

などと複雑な条件式などでは，フリーフォーマットである事を積極的に利用して，

```
if ( (kansuu1 () && flag1) ||
     (kansuu2 () && flag2) )
```

のように条件を複数の行に分けて記述することができます．このように記述すると単純にコンパイルエラーの発生を防げるだけでなく，ロジックがわかりやすくなるという特徴もありますから，試してみると良いでしょう．

(3) タイプミス

if，forなどは短いのでミススペルをすることはまずありませんが，while，switch，defaultなどは誤りやすいので気をつけてください．誤って覚えていたりすると意外と悩むものです．

(4) 文字列のダブルクォーテーションマーク

文字列は必ず“　”で囲まなければなりませんが，最後の”を忘れることがあります．これは通常

```
String too long (or missing quote)
```

というメッセージが出るため，すぐに修正することができますが，＃if，＃endifなどで条件コンパイルを指定しているとエラー発生行を誤って表示する場合があります．

リスト3-1-2　string too longエラーがでない場合

```
1: #include <bdscio.h>
2:
3: #define DEBUG   0
4:
5: main()
```

```
 6: {
 7:         printf ("it is in main().);
 8:         test ();
 9: }
10:
11: test()
12: {
13: #if DEBUG
14:         printf ("it is in test()");
15: #endif
16: }
```

ここに文字列の終了を示すダブルクォーテーション”が落ちている．

〈コンパイル例〉

```
A>cc f:error2
BD Software C Compiler v1.50a (part I)
  13: Conditional expr bad or beyond implemented subset

A>
```

このエラーメッセージは13行めの＃ifの使い方が正しくないという指摘．
しかし，それは正しくない．このエラーは7行めの”不足のため引き起こされたもの．
このエラーメッセージはα-Cでも同様に出現する．

10行めで落した”が100行めの＃ifでエラーを発生させた場合など，非常に苦しむことになります．もし＃ifで発生したエラーがあったら，文字列の”を落していないかチェックしてください．

コンパイルエラーは最初は非常に多いものですが，慣れるとその発生頻度も減ってきますし，修正の要領も判ってきます．99%のエラーはそのメッセージによってどういうものか判定できますから，少しずつエラーを減らすようにしていけば良いでしょう．

3-2　BDS Cで削除されている機能

BDS Cおよびα-CはC言語の仕様を一部削除したサブセット版です．通常の使用ではあまり問題になることはないのですが，本質的な仕様の点で次

のような制約がありますから，注意しなければなりません．

(1) 実数（float），倍精度整数（long），倍精度実数（double）が使えない．

(2) スタティック変数がない．

(3) 変数の初期値設定機能（イニシャライザ）がない．

BDS Cでは実数，倍精度整数については関数パッケージ(float，long）が用意されており，関数の形で用いることができます(α-Cには付属していません)．しかし，他のCコンパイラ用に作成されたプログラムなどをこの関数を用いてBDS Cに移植するのはかなり大変な作業です．float関数の具体的な使い方については，3-4項をご覧ください．

(2)，(3)はC言語の本質的な部分であり，非常に残念なことです．BDS Cは完成されたオブジェクトをROMなどに焼き込み，制御機器などへの組み込み用ソフトウェアを作成することができますから，この機能を付加することが難しく，削除されているのだと思います．

しばしば，C言語のサンプルとして，次のようなものがあります．

リスト3-2-1　イニシャライザを用いたプログラムとコンパイル例

```
1: #include <bdscio.h>
2:
3: main()
4: {
5:             int        a = 100;
6:             static int b = 200;
7:             printf ("int a=%d¥n",a);
8:             printf ("static int b=%d¥n",b);
9: }
```

intのデフォルトが自動変数（auto）になっている場合，どのようなコンパイラでもこの行はエラーになる．

スタティック変数と明示されているため，一般のCコンパイラでは初期化機能が働く．

<コンパイル例>

```
A>cc init1
BD Software C Compiler v1.50a (part I)
```

```
    5: Missing semicolon
    6: Undeclared identifier: static ←── staticはBDS Cではサポートされていない.
    6: Missing semicolon

A>
```

非常に簡単なプログラムですし，他のコンパイラでは大抵コンパイルすることができますが，上記のようにエラーがでてしまいます．このような場合は，ローカル変数，あるいは外部変数を用いて変数を定義し，プログラム先頭でその変数に初期値を設定するようにします．

リスト3-2-2 初期値の設定プログラム

```
 1: #include <bdscio.h>
 2:
 3: main()
 4: {
 5:         int     a;  ┐
 6:         int     b;  ┘←── 変数は必ず宣言だけ行なう.
 7:         a = 100;  ┐
 8:         b = 200;  ┘←── 変数へは値を代入するようにすると
                            (意味あいは異なるが)初期化が達成される.
 9:         printf ("int a=%d¥n",a);
10:         printf ("int b=%d¥n",b);
11: }
```

```
A>cc init2
BD Software C Compiler v1.50a (part I)
  35K elbowroom
BD Software C Compiler v1.50 (part II)
  32K to spare

A>clink init2
BD Software C Linker   v1.50
Linkage complete
  43K left over

A>init2
```

```
int a=100
int b=200

A>
```

この方法は実行時間がかかる，オブジェクトのサイズが大きくなるという点でエレガントな方法ではありません．しかし，他のCコンパイラにかけても問題ありませんし，ローカル変数に初期値を設定し忘れることも無くなります．

それより，問題は大きな配列などに初期値を設定したい場合です．リスト3-2-3は0から7をデコードし，1～0x80を戻り値として返す関数の例ですが，スタティック変数の初期値を設定可能であれば，次のようにプログラムすることができます．

リスト3-2-3 配列を初期化するプログラム

```
 1: #include <bdscio.h>
 2:
 3: main()
 4: {
 5:         int     i;
 6:
 7:         for (i=0; i<8; i++)
 8:                 printf ("%4d  %04x¥n",i,decode(i));
 9: }
10:
11:
12: decode(n)
13:         int     n;
14: {
15:         static int array[]={1,2,4,8,16,32,64,128};
16:         return (array[n]);
17: }
```

```
A>cc init3
BD Software C Compiler v1.50a (part I)
```

```
15: Undeclared identifier: static←――static変数はサポートされていない.
15: Need explicit dimension size←―配列宣言では[ ]の中に必ず要素の数が必要.
16: Undeclared identifier: array←――15行めのエラーでarrayが登録さ
                                    れなかったため，連鎖的に発生し
                                    たエラー.

A>
```

この場合は高々8通りですから，switch文とcase文を用いてもプログラミングできますが，もっと多くなってくるとそうもいきません．そこで，プログラムの実行開始時に外部変数に値を書き込み，関数中でその配列を参照するような方法をとることもできます．その例をリスト3-2-4に示します．

リスト3-2-4　配列に外部変数を用いた例

```
 1: #include <bdscio.h>
 2:
 3: int     array[16];←――外部変数として配列を宣言
 4:
 5: main()
 6: {
 7:         int     i;
 8:         setarray ();←――配列に値を代入しておく
 9:
10:         for (i=0; i<8; i++)
11:                 printf ("%4d  %04x¥n",i,decode(i));
12: }
13:
14:
15: decode(n)
16:         int     n;
17: {
18:         return (array[n]);
19: }
20:
21:
22: setarray()←――配列に値を代入する関数
23: {
24:         array[0]=1;
25:         array[1]=2;
26:         array[2]=4;
27:         array[3]=8;
28:         array[4]=16;
```

```
29:            array[5]=32;
30:            array[6]=64;
31:            array[7]=128;
32: }


A>cc init4
BD Software C Compiler v1.50a (part I)
  35K elbowroom
BD Software C Compiler v1.50 (part II)
  32K to spare

A>clink init4
BD Software C Linker   v1.50
Linkage complete
  43K left over


A>init4

   0  0001
   1  0002
   2  0004
   3  0008
   4  0010
   5  0020
   6  0040
   7  0080

A>
```

実際BDS Cで本格的なプログラムを作成する場合，最大のネックになるのはこの初期値の設定で，場合によってはアセンブラを使った方が良い時もあります（アセンブラによるテーブルピックアップはプログラムとしては非常に簡単です）．

また，intなどの整数は，この方法でも十分ですが，文字列（文字配列）の初期化はなかなか難しい問題です．マニュアルにはstrcpy関数を用いるとありますが，文字列が多い場合には無駄が多いので，賛成しかねます．それよりも，プログラムの構造自体に関わりますが，あらかじめポインタ配列を用

意しておき，最初にその配列に文字列の先頭アドレスをセットするようにしておくと文字列はプログラムの一部として埋め込まれるので効率的だと思います．ただし，この方法は文字列への書き込みができません（元の文字列より短ければ不可能ではありません）．

リスト3-2-5　文字列へのポインタ配列を作る例

```
 1: #include <bdscio.h>
 2:
 3: chaı       *array[3];←――――――文字列へのポインタ配列の宣言.
 4:
 5: main()
 6: {
 7:            int     i;
 8:            setarray ();←――――――ポインタ配列に文字列の
                                    先頭アドレスをセットする.
 9:            for (i=0; i<3; i++)
10:                    printf ("string %d=%s¥n",i,array[i]);
11: }
12:
13: setarray()
14: {
15:            array[0]="Good morning.";
16:            array[1]="Good afternoon.";
17:            array[2]="Good night.";
18: }
```

```
A>cc init5
BD Software C Compiler v1.50a (part I)
  35K elbowroom
BD Software C Compiler v1.50 (part II)
  32K to spare


A>clink init5
BD Software C Linker   v1.50
Linkage complete
  43K left over


A>init5
```

```
string 0=Good morning.
string 1=Good afternoon.
string 2=Good night.

A>
```

いずれにせよ，本来のC言語では悩む必要も無い部分なので，将来の機能拡張を望みたいところです．

3-3 プリプロセッサの機能

プリプロセッサとはプログラムをコンパイルする前に文字列の置き換えを行なったり，条件コンパイルなどによるソースリストの削除を行なう機能で，C言語を使いこなす上で，非常に重要なポイントの1つです．

BDS C，α-Cのプリプロセッサには次のようなものがあります．

```
#define
#undef
#include
#if
#ifdef
#ifdef
#else
#endif
```

これらのうち，#defineは文字列の置き換え，#includeはファイルの取り込み，それ以降は条件コンパイルのために用いられます．上記以外のプリプロセッサはBDS Cでは削除されています．また，プリプロセッサではありませんが，コメントもコンパイラの重要な機能です．

(1) #define

#defineは次のような書式で用いられます．

#define　　定義名　定義値

一番多い使い方は次のように文字列に数値を設定する例です．

リスト3-3-1　数値展開の例

```
#include <bdscio.h>

#define MAXNUM  10

main()
{
        int     i;
        for (i=0; i<MAXNUM; i++)
                printf ("%d¥n",i);
}


A>define1
0
1
2
3
4
5
6
7
8
9

A>
```

定義名と定義値の間はスペースが区切りであるため，スペースを含む文字列を定義名として指定することはできません．また，この方法で指定した数値は数値をシンボル名で参照するわけではなく，

　　"MAXNUM"　→　"10"

と別の文字列に置き換えるだけです．ここでMAXNUMは大文字になっていますが，C言語では定数は大文字で記述するのが一般的で，小文字でも別に動作には問題ありませんが，インデント（字下げ）と共にC言語の標準プログラミングスタイルの1つです．

次に，#defineは，単純な文字列置き換えだけでなく，引数を持ったマクロ展開も行なうことができます．

リスト3-3-2 マクロ展開の例

```
#include <bdscio.h>

#define lowcase(x) (x+('a'-'A'))

main()
{
        char    *s;

        s = "ABCDEFGHIJKLMNOPQRSTUVWXYZ";
        while (*s)      putchar (lowcase(*s++));
}


A>define2
abcdefghijklmnopqrstuvwxyz

A>
```

<注意>
これは例なので，このマクロ定義では大文字以外の文字がxに設定された時には問題が発生する．

大文字が小文字に変換される．

マクロ展開と言えども単なる文字列置き換えに過ぎないということには注意する必要があります．数値以外に関数呼出しや演算式を書く可能性がある部分には，引数自体を（ ）で囲み，評価の順序などが誤らないようにする必要があります．リスト3-3-3は（ ）を付けなかったため，誤った答となる例です．

リスト3-3-3　誤ったマクロ展開の例

```
#define minus(n) -n

main()
{
        printf ("%d",minus(10+2));
}



A>cc define3 -p
BD Software C Compiler v1.50a (part I)

   1:
   3: main()
   4: {
   5:   printf ("%d",-10+2);
   6: }              ↑──minus(10+2)がこのように
                        展開されてしまう.
  36K elbowroom
BD Software C Compiler v1.50 (part II)
  32K to spare

A>clink define3
BD Software C Linker   v1.50
Linkage complete
  43K left over


A>define3
-8←────────答は誤り.

A>
```

コンパイル時の－pはプリプロセッサを通ったソースファイルがどのように変化したかを確認するために設けられているオプションで，例のように実際に展開された様子が画面に出力されます．この例では，

```
minus(10+2)
    ↓
 -10+2
```

と展開されてしまいます．－(10＋2) としたいわけですから，次のように修正しなければなりません．

リスト3-3-4　正しいマクロ展開の例

```
#define minus(n) -(n)←──── nをカッコで囲んでいる点に注意．

main()
{
        printf ("%d",minus(10+2));
}


A>cc define4 -p
BD Software C Compiler v1.50a (part I)

   1:
   3: main()
   4: {
   5:   printf ("%d",-(10+2));
   6: }           └──── ( )つきで展開されている．

  36K elbowroom
BD Software C Compiler v1.50 (part II)
  32K to spare

A>clink define4
BD Software C Linker   v1.50
b:Linkage complete
  43K left over


A>define4
-12←──── 答は正しい．

A>
```

慣れるまで時間がかかるとは思いますが，リスト3-3-3のような例は見落しがちなので，注意してください．

(2) ＃undef

＃undefは＃defineによる定義を抹消する場合に用います．

＃defineにより同じ名前を定義すると後のものが優先されますが，入れ子になるわけではなく，＃undefすると前の定義は残りません．

リスト3-3-5 ＃undefの例

```
main()
{
#define DEF     10
        printf ("DEF1=%d¥n",DEF);
#define DEF     100
        printf ("DEF2=%d¥n",DEF);
#undef  DEF
        printf ("DEF3=%d¥n",DEF);
}


A>CC UNDEF -P
BD Software C Compiler v1.50a (part I)

   1: main()
   2: {
   3:
   4:   printf ("DEF1=%d¥n",10);
   5:
   6:   printf ("DEF2=%d¥n",100);
   7:
   8:   printf ("DEF3=%d¥n",DEF);
   9: }

   8: Undeclared identifier: DEF

A>
```

最後に定義された内容が有効になる．

定義が抹消されているためエラーとなる．

このようなプログラムを作ることはあまりありませんが，知っておくと便利なこともあるでしょう．

(3) ＃include

＃includeはファイルの取り込みを指定しますほとんどのBDS Cおよびα-Cのプログラムでは，プログラム先頭に

```
#include <bdscio.h>
```

というヘッダーファイルの取り込みを記述します．この中にはいくつかの外部変数の宣言，構造体などの定義が記述されており，BDS Cで用意されている標準関数を使う場合には必ず指定する必要があります．

このbdscio.hは一般のC言語ではstdio.h(Standard Input Output，標準入出力）という名前になっている場合が多いので，他のC言語に移植する場合には変更する必要があります．

次に，＃includeでは，書式に2種類あり，

```
#include <filename>
#include "filename"
```

の両者があります．ファイル名を＜＞で囲んだ場合にはログインディスク(プロンプトがA＞の場合はドライブA）から捜され，“ ”の場合はソースファイルと同じディスクから捜されます．実際のプログラム開発の場合，コンパイラ本体はドライブAに置いておくことが多いのでbdscio.hなど共通のヘッダーファイルも一緒にして＜＞で囲み，それ以外ユーザー側で作成したヘッダーファイルなどはソースファイルなどと一緒に異なるディスクに置き，“ ”で囲むようにすると良いでしょう．

なお，〈 〉，“ ”の双方ともファイル名にドライブ(A：，B：など）を記述するとそちらが優先されます．

目的のファイルが見つからないとそこでコンパイルは中断されます．

(4) 条件コンパイル

＃if, ＃ifdef, ＃ifndef, ＃else, ＃endif はすべて条件コンパイルを指定するプリプロセッサですが，次のように用います．

```
#if    式
   ⋮
#else
   ⋮
#endif
```

あるいは

```
#ifdef    識別子
   ⋮
#else
   ⋮
#endif
```

あるいは

```
#ifndef   識別子
   ⋮
#else
   ⋮
#endif
```

＃if の場合は式の値が真（0以外）であれば＃else までをコンパイルし，それ以降＃endif までをスキップします．偽(0) なら＃else 以降＃endif までをコンパイルします．＃ifdef, ＃ifndef もほとんど同じです．＃ifdef では識別子(名前)が＃define によって定義されていれば，＃else までをコンパイルし，＃endif までをスキップします．＃ifndef は＃ifdef の逆になります．

いずれの場合も＃elseは省略することが可能です．また，これらの指定はネストすることができます．

リスト3-3-6　条件コンパイルネストの例

```
#define TEST1    1

main()
{
#if TEST1
        printf ("#if TEST1 compiled.¥n");
 #ifdef TEST2←――――TEST2は定義されていない.
        printf ("#ifdef TEST2 compiled.¥n");
 #else
        printf ("TEST2 #else compiled.¥n");
 #endif
#else
        printf ("TEST1 #else compiled.¥n");
#endif
}


A>cc ifdef -p
BD Software C Compiler v1.50a (part I)

   1:
   3: main()
   4: {
   5:
   6:   printf ("#if TEST1 compiled.¥n");←
   7:                                        この2つだけが
  10:   printf ("TEST2 #else compiled.¥n");←  有効となる.
  11:
  15: }

  36K elbowroom
BD Software C Compiler v1.50 (part II)
  32K to spare
```

```
A>clink ifdef
BD Software C Linker   v1.50
Linkage complete
  43K left over


A>ifdef

#if TEST1 compiled.
TEST2 #else compiled.

A>
```

なお，この条件コンパイルはデバッグ時などデバッグ用プログラム部分を指定しておき，プログラム完成時に設定を変えるだけで削除したい場合に用います．

(5) **コメント**

コメントは文字列/*から*/までをコメントとみなし，コンパイル時に無視します．コメントはネストすることが可能で，

```
        /*
              /*    comment    */
                                      */
```

という形式で使うこともできます．

3-4　浮動小数点演算

整数演算しかできないBDS Cにとって，小数点以下の値を含んだ数や，あるいは大きな数を扱う事のできる浮動小数点数演算パッケージ(float)は魅力あるものです．これらはうまく使うと非常に効果的にプログラムを作成す

ることができます．

しかし，関数の形でしか利用できませんから，互換性の点では他のCコンパイラにかける可能性がある場合には問題外となります．あくまでBDS Cで実数を使いたい方が用いるものと考えるべきでしょう．

しかし，数個の実数計算がある，あるいは平均点の結果だけ小数点表示を行ないたいといった場合にはかなり救われることも事実で，恩恵は少ないとは言えません．

さて，float関数を説明する前に，一言コメントしておきます．

float関数にはオーバーフロー・アンダーフローを抑止する機能がありません．一般には演算を行なって扱える数値の範囲を越えた場合，計算を打ち切って，扱える数のうち最もその値に近い値を返すものが多いのですが，float関数の場合，全くデタラメな値を返してきます．これは致命的なバグです．

一般には，非常に小さい値，あるいは大きな値を扱う可能性がある場合には，適当な時点で計算を打ち切り，オーバーフロー，アンダーフローが起らないような形でプログラミングする必要があります．ちなみに，扱える値は絶対値が約0.340282e39～0.293874e－38の範囲の値と0です．

これらのパッケージはBDS Cの開発者であるゾールマン氏の作成ではないことも，メンテナンスの点で問題を残しているようです．α-Cがfloat，longパッケージを含んでいないのもそんな所に原因があるのかも知れません．

(1) float関数の使い方

floatの数値はBDS Cでは5バイトのchar型変数配列で表されます．例えば，2つの実数の加算を行なう場合，リスト3-4-1のようにします．

リスト3-4-1　実数の加算と操作例

```
#include <bdscio.h>

main()
{
        char    a[5];  )
        char    b[5];  } 実数は5バイトのchar型配列で表わされる．
        char    c[5];  )

        atof (a,"1.23");←―――――a=1.23
        itof (b,1);←―――――――b=1
        fpadd (c,a,b);←―――――c=a+b
        printf ("%f+%f=%f",a,b,c);
}


A>cc ftest1
BD Software C Compiler v1.50a (part I)
  35K elbowroom
BD Software C Compiler v1.50 (part II)
  32K to spare

A>clink ftest1 -f float
BD Software C Linker   v1.50
Linkage complete
  39K left over


A>ftest1

1.230000+1.000000=2.230000

A>
```

ここで，

atof	文字列を実数に直す関数
itof	int型の数値を実数に直す関数
fpadd	実数の加算をする関数

です．BDS Cでは実数は5バイトの文字配列でしか表わすことができず，じかに数値を記述したりすることはできません．定数であろうとも，必ず文字配列を用意し，atof，itofなどの関数を用いて値をセットする必要があります．また，リンク時には例のように，必ず

```
clink filename −f float
```

とし，リンクファイルの後に－fオプションを付加してください．これがないと正しいリンクを行なうことができません．特に，floatパッケージの中にはfloat版のprintf関数が含まれており，このようにしないと通常のprintf関数が誤ってリンクされてしまうことになります．この場合，例のように％fを用いて実数を画面に表示することができなくなりますので，注意してください．

なお，一般の整数では

```
c=a+b;
```

とでも記述すれば良いプログラムが実数を用いた場合には例のようにかなり面倒な処理を行なわなくてはならなくなります．非常に良く間違うポイントですので，十分注意してください．

なお，floatパッケージの関数の戻り値は演算結果を格納したバッファーのアドレスです．従って，次のような使い方は可能です．

リスト3-4-2 関数の戻り値をそのまま使う例

```
#include <bdscio.h>

main()
{
        char    a[5];
        char    b[5];
        char    c[5];
```

```
        atof (a,"1.23");
        itof (b,1);
        printf ("%f+%f=%f",a,b,fpadd(c,a,b));
}
```

関数の戻り値をそのまま利用する．

```
A>cc ftest2
BD Software C Compiler v1.50a (part I)
  35K elbowroom
BD Software C Compiler v1.50 (part II)
  32K to spare

A>clink ftest2 -f float
BD Software C Linker   v1.50
Linkage complete
  39K left over


A>ftest2

1.230000+1.000000=2.230000

A>
```

このように，関数の戻り値をそのまま使うと，一般の演算の場合のように表現できるので，リストが短く，見やすくなります．また，floatパッケージの内部では実数演算用の専用のメモリを確保しており，いったんそこへ値をコピーしてから演算しますので，入力と出力に同じ文字列を指定することも可能です．（そのかわり，ROM化などの用途には使えません．）

リスト3-4-3　入力と出力に同じ文字列を指定した例

```
#include <bdscio.h>

main()
{
        char    a[5];
        char    b[5];
```

```
        atof (a,"1.23");
        itof (b,1);
        printf ("%f+%f=",a,b);
        printf ("%f",fpadd(a,a,b));
}
```

入力と出力に同じ文字配列が指定されている．

```
A>cc ftest3
BD Software C Compiler v1.50a (part I)
  35K elbowroom
BD Software C Compiler v1.50 (part II)
  32K to spare

A>clink ftest3 -f float
BD Software C Linker   v1.50
Linkage complete
  39K left over

A>ftest3

1.230000+1.000000=2.230000

A>
```

ただし，ここで注意しなければならないことは，実数aはfpadd(a, a, b)の演算によりa＋bの内容に置き換えられてしまうことです．つまり，

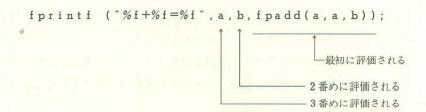
ｆｐａｄｄ（ａ，ａ，ｂ）→　ａ＝ａ＋ｂ

であることを念頭におくべきです．さもないと次のような誤りを犯すことになります．

リスト３-４-４　fpaddの使う位置を誤った例

```
#include <bdscio.h>

main()
{
```

```
        char    a[5];
        char    b[5];

        atof (a,"1.23");
        itof (b,1);
        printf ("%f+%f=%f",a,b,fpadd(a,a,b));←この行に誤りがある.
}


A>cc ftest4
BD Software C Compiler v1.50a (part I)
  35K elbowroom
BD Software C Compiler v1.50 (part II)
  32K to spare

A>clink ftest4 -f float
BD Software C Linker   v1.50
Linkage complete
  39K left over


A>ftest4

2.230000+1.000000=2.230000
 ↑
 └──── aの値はすでに2.23になっている.
A>
```

BDS C，α-C では，関数への引数は最後から評価されます.

従って，この例ではfpaddの関数が先に実行され，aの値がa＋bに変化してしまってから最初に記述されているaを評価することになるため，例3-4

-4のようにおかしなことになるわけです．例3-4-3のようにprintf文を分割するか，別の配列に出力しなければなりません．

なお，関数への引数が複数ある場合，どういう順序で評価するかはC言語仕様上定まっていません．BDS Cでは最後から行ないますが，他のコンパイラでは異なる順番で評価する場合があるため，このように引数の評価順序に依存するプログラミングは好ましくありません．

(2) fplong.h

float関数を使う場合，実数が5文字の文字列であるということはかなり大きな制約となります．プログラム中でも，

```
function( )
{
    char a[5];
        ⋮
```

という形で宣言すると，この文字配列aは果して通常の文字列として用いるのか，実数として用いるのか判りにくくなります．そこで，次のような形で，実数が宣言できると便利です．

```
float a;
float k,l;
```

このため，実数を5文字の文字列から構成される構造体で表し，その構造体名をfloatとするヘッダーファイルを用意しました．このファイルでは同時にいくつかの関数についてマクロ定義を行なっているので実行がやや高速になる他，倍精度整数を用いるlongパッケージの関数についても同様な定義を行なっていますので，

```
_long a;
_long k,l;
```

という形で倍精度整数が宣言できます．longは関数名と同じなので，使えないため，__longになっています．

リスト3-4-5　fplong.h

```
/*      float macro function        */

#define fpnorm(op1)             fp(0,op1,op1)                              正規化
#define fpadd(r,op1,op2)        (fp(1,r,op1,op2),r)  /* r <- op1+op2 */    加算
#define fpsub(r,op1,op2)        (fp(2,r,op1,op2),r)  /* r <- op1-op2 */    減算
#define fpmult(r,op1,op2)       (fp(3,r,op1,op2),r)  /* r <- op1*op2 */    乗算
#define fpdiv(r,op1,op2)        (fp(4,r,op1,op2),r)  /* r <- op1/op2 */    除算
#define ftoa(r,op)              (fp(5,r,op),r)       /* float to ascii */  実数→文字列変換

/*      long macro function         */

#define itol(r,op)              long(0,r,op)         /* int to long  */
#define lcomp(op1,op2)          long(1,op1,op2)      /* if (op1==op2) 0    比較
                                                        op1>op2 ? 1:-1; */
#define ladd(r,op1,op2)         long(2,r,op1,op2)    /* r <- op1+op2 */    加算
#define lsub(r,op1,op2)         long(3,r,op1,op2)    /* r <- op1-op2 */    減算
#define lmul(r,op1,op2)         long(4,r,op1,op2)    /* r <- op1*op2 */    乗算
#define ldiv(r,op1,op2)         long(5,r,op1,op2)    /* r <- op1/op2 */    減算
#define lmod(r,op1,op2)         long(6,r,op1,op2)    /* r <- op1%op2 */    剰余

struct  _float
        {
                char    _fpdim[5];
        };
#define float   struct  _float

struct  _lg
        {
                char    _lgdim[4];
        };
#define _long   struct  _lg
```

なお，このヘッダーファイルを用いた場合，引数については次のようにポインタで宣言します．

```
function (a)
  float *a;
{
        ⋮
```

また，fplong.hを用いるもう一つのメリットは，この中で定義されている関数に限り，α-Cでも用いることができるということです．これは，float，long

関数の本体部分は標準関数としてdeff2. crlの中に格納されているためです．

このヘッダーファイルの使用例については以降のサンプルを参考にしてください．

(3)　三角関数

sin関数は非常に一般的に用いられる関数ですが，マクローリン展開

式3-4-6

$$\sin x=\sum_{n=0}^{\infty}(-1)^n\frac{x^{2n+1}}{(2n+1)!}$$
$$=x-\frac{x^3}{3!}+\frac{x^5}{5!}-\frac{x^7}{7!}+\cdots\cdots$$

を用いる事により，容易に関数値を求めることができます．ただ，このような式で近似値を求めるのはfloatパッケージにはかなり荷が重いようです．特に，項の値が小さい場合，アンダーフローが起りやすく，プログラミングにはかなり気を使います．

ここで紹介するsin関数はある程度以上値が小さくなるとそこで計算を打ち切り，アンダーフローを防ぐようにしています．

なお，ここでは前述のfplong. hのヘッダーファイルを用いています．

```
char *sin(put,deg)
  char *pnt;
  int  deg;
```

pntには求めた値を格納する5バイトの文字配列の先頭アドレスを指定します．degは角度ですが，ここで与える値は実際の角度に10を掛けたものです．すなわち，この関数では0.1度単位で指定できるということになります．これはリストの28行めで設定されているTENの値を10から100に直せば，0.01度単位で利用できるsin関数になりますが，整数は－32768～32767の値しか

取れないので，−327.68度から327.67度までしか使えないsin関数となります．適当にこの値を変更してください．

リスト3-4-7は10度単位にsin関数値を表示するプログラムです．結果を見れば判るように単精度ですので余り精度は高くありません．なお，展開式の10項までで計算を打ち切るようになっていますが，演算結果がある値（CALCMAX）より小さくなると計算を打ち切ります．

リスト3-4-7 sin.cと実行例

```
 1:   #include <bdscio.h>
 2:   #include "fplong.h"
 3:   #define  CNTMAX 10
 4:   main()
 5:   {
 6:           float   s,t;
 7:           int     i;
 8:
 9:           printf ("SIN      VALUE:¥n");
10:           printf ("angle     sin¥n");
11:           for (i=0; i<=1800; i+=100)
12:           {
13:                   sin (t,i);
14:                   printf("%3d %15.10f¥n",i/10,t);
15:
16:           }
17:   }
18:
19:   sin(r,data)
20:           float   *r;
21:           int     data;
22:   {
23:           int     i,isign;
24:           float   s,x,b,e,f,g,xx,h,k,sign,xy;
25:           float   ONE,RD,TEN,CALCMAX,RDD;
26:
27:   /* Set values   */
28:           itof (TEN,10);
29:           atof(RD,"0.3046e-3");   /* RDD*RDD */
30:           itof(ONE,1);
31:           atof(CALCMAX,"1e-30");
32:
```

これらは変数だが実際には定数として扱っている．それを明示するために大文字で記述してある．

```
33:         atof(RDD,"0.0174533"); /* 3.1415927/180 */
34:
35:   /* Variables initialize */
36:         isign = -1;
37:         itof (sign,isign);
38:         itof (x,data);
39:         fpdiv (x,x,TEN);
40:         fcopy (xy,x);
41:
42:         fpmult(s,xy,RDD);       /* s is result  */
43:         fcopy (k,s);
44:
45:         itof(e,1);              /* kaijo variable       */
46:         itof(g,1);              /* kaijo result         */
47:         itof(f,1);
48:
49:         fpmult (xx,x,x);
50:         fpmult (xx,xx,RD);      /* xx = x*x*RD          */
51:
52:   /* Calculation  */
53:         for (i=2;i<CNTMAX;i++)
54:         {
55:                 fpadd  (e,e,ONE);       /* e=e+1        */
56:                 fcopy  (g,e);           /* g=e          */
57:                 fpadd  (e,e,ONE);       /* e=e+1        */
58:                 fpmult (g,g,e);         /* g=g*e        */
59:
60:                 fpmult (h,xx,k);        /* h=xx*k       */
61:                 fpdiv  (k,h,g);         /* k=h*g        */
62:
63:                 if (fpcomp (k,CALCMAX) == -1)   break;
64:
65:                 fpmult (f,k,sign);
66:                 fpadd  (s,s,f);         /* s=s+f        */
67:                 isign = -isign;
68:                 itof (sign,isign);
69:         }
70:
71:
72:         fcopy (r,s);
73:
74:   }
75:
76:   fcopy(t,o)
77:         char    *t,*o;
```

```
78:   {
79:           int     i;
80:           for (i=0; i<5; i++)      *t++=*o++;
81:   }
82:
```

〈実行例〉

```
A>sin

SIN     VALUE:
angle    sin
  0    0.0000000000
 10    0.1736483000
 20    0.3420208000
 30    0.5000017000
 40    0.6427912000
 50    0.7660509000
 60    0.8660359000
 70    0.9397083000
 80    0.9848297000
 90    1.0000290000
100    0.9848453000
110    0.9397389000
120    0.8660807000
130    0.7661086000
140    0.6428597000
150    0.5000794000
160    0.3421047000
170    0.1737365000
180    0.0000897514

A>
```

リストの中で用いられているfcopyという関数は実数へ値を代入する関数です.

```
fcopy(a,b)
  float *a,*b;
```

で，実数bを実数aに代入します.

```
char *cos(pnt,deg)
  char *pnt;
  int  deg;
```

sin関数と使いかたは全く同じです．こちらも同じくマクローリン展開式，

式3-4-8

$$\cos x=\sum_{n=0}^{\infty}(-1)^n\frac{x^{2n}}{(2n)!}$$
$$=1-\frac{x^2}{2!}+\frac{x^4}{4!}-\frac{x^6}{6!}+\cdots$$

を用いていますが，一般には，sin関数，cos関数のどちらかが判れば，

$$\sin x=\cos(x-900)$$
$$\cos x=\sin(x+900)$$

を利用すれば良いでしょう．リスト3-4-9はcos関数値を10度単位で表示するプログラムです．

リスト3-4-9　cos.cと実行例

```
 1:  #include <bdscio.h>
 2:  #include "fplong.h"
 3:  #define  CNTMAX 10
 4:
 5:  main()
 6:  {
 7:          float   s;
 8:          int     i;
 9:
10:          printf ("COS VALUE:¥n");
11:          printf ("angle    cos¥n");
12:          for (i=0; i<=1800; i+=100)
13:          {
14:                  cos (s,i);
```

```
15:                     printf("%3d %15.10f¥n",i/10,s);
16:             }
17:     }
18:
19:     cos(r,data)
20:             float   *r;
21:             int     data;
22:     {
23:             int     i,isign;
24:             float   s,x,b,e,f,g,xx,h,k,sign;
25:             float   ONE,RD,TEN,CALCMAX;
26:
27:     /* Set values   */
28:             itof (TEN,10);
29:             atof(RD,"0.3046e-3");    /* RDD*RDD */
30:             itof(ONE,1);
31:             atof(CALCMAX,"1e-30");
32:
33:     /* Variables initialize */
34:             isign = -1;
35:             itof (sign,isign);
36:             itof (x,data);
37:             fpdiv (x,x,TEN);
38:
39:             itof(s,1);               /* s is result  */
40:             itof(e,0);               /* kaijo variable       */
41:             itof(g,1);               /* kaijo result         */
42:             itof(f,1);
43:             itof(k,1);
44:
45:
46:             fpmult (xx,x,x);
47:             fpmult (xx,xx,RD);       /* xx = x*x*RD          */
48:
49:     /* Calculation  */
50:             for (i=2;i<CNTMAX;i++)
51:             {
52:                     fpadd  (e,e,ONE);        /* e=e+1        */
53:                     fcopy  (g,e);            /* g=e          */
54:                     fpadd  (e,e,ONE);        /* e=e+1        */
55:                     fpmult (g,g,e);          /* g=g*e        */
56:
57:                     fpmult (h,xx,k);         /* h=xx*k       */
58:                     fpdiv  (k,h,g);          /* k=h*g        */
59:
```

```
60:                     if (fpcomp (k,CALCMAX) == -1)    break;
61:
62:                     fpmult (f,k,sign);
63:                     fpadd  (s,s,f);          /* s=s+f        */
64:                     isign = -isign;
65:                     itof (sign,isign);
66:             }
67:             fcopy (r,s);
68:
69:     }
70:
71:     fcopy(t,o)
72:             char    *t,*o;
73:     {
74:             int     i;
75:             for (i=0; i<5; i++)     *t++=*o++;
76:     }
77:
```

〈実行例〉

```
A>cos

COS VALUE:
angle    cos
  0    1.0000000000
 10    0.9848086000
 20    0.9396960000
 30    0.8660328000
 40    0.7660572000
 50    0.6428066000
 60    0.5000258000
 70    0.3420528000
 80    0.1736874000
 90    0.0000448172
100   -0.1735991000
110   -0.3419686000
120   -0.4999482000
130   -0.6427380000
140   -0.7659996000
150   -0.8659878000
160   -0.9396652000
170   -0.9847926000
180   -0.9999997000

A>
```

さて，sin，cos 関数があれば tan 関数は

$$\tan\ x=\frac{\sin\ x}{\cos\ x}$$

で求めるのが簡単ですが，sin，cos 関数と異なり，tan(90°)は∞となるため，ある値以上では実数がとりうる最大の値を返すように工夫しなければなりません．

以上，三角関数のサンプルを紹介しましたが，述べたように，アンダーフロー，オーバーフローする場合があるために，完全なプログラムを作ることはできません．プログラムを変更した場合には誤った値を発生してしまう可能性があることに注意してください．また，これらの関数はアルゴリズムの問題もあり，精度は4桁程度で処理速度も遅くなっています．実際に利用される場合にはあらかじめ許容できるかどうか確認してください．

なお，最後に申し添えておきますが，信頼性の高いソフトウェアを作成する場合，実数演算を必要とするなら，BDS C を用いるにはかなり工夫する必要があります．

第 4 章

CP/Mサンプルプログラム

BDS C，α-Cで作成されるプログラムで最も多いものはなんといってもCP/M用のユーティリティプログラムです．あれば便利といったものから，言語，高度なアプリケーションに至るまで，多くがBDS Cで作成されています．

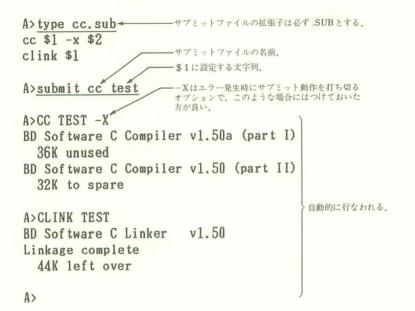
本章では，CP/Mに付属しているSUBMITを拡張したexsub(expanded submit）プログラムを紹介し，説明します．

4-1　拡張サブミットコマンド@.com

CP/Mにはバッチ処理用のコマンドとして，SUBMITが用意されており，これにより，一連の作業を自動化することができます．

例えば，test.cというプログラムを開発する場合，あらかじめバッチ処理の順番を記述したファイル(.SUBファイル)を作っておくと，SUBMITコマンドにより，自動的にcomファイルまで作成することが可能になります．

リスト4-1-1　SUBMITファイルの例

```
A>type cc.sub
cc $1 -x $2
clink $1

A>submit cc test

A>CC TEST -X
BD Software C Compiler v1.50a (part I)
  36K unused
BD Software C Compiler v1.50 (part II)
  32K to spare

A>CLINK TEST
BD Software C Linker   v1.50
Linkage complete
  44K left over

A>
```


また，SUBMITの実行を助けるXSUBというコマンドもサポートされており，プログラム中のキーボード入力をSUBファイルから行なうことも可能になっています．このように，SUBMITは便利なコマンドですが，いくつかの欠点があり，使いづらい点があります．

そこで，より使いやすさを考慮した拡張SUBMITコマンドを作成しましたのでご紹介したいと思います．このプログラムはSUBMITを包含していま

すので，今迄のSUBファイルもそのまま利用することができます．

完成したexsub.comは@.comという名前にリネームしておくと，タイピングが楽になります．また，exsubをサポートするサブプログラム/.comを利用するとさらに有効に利用することが可能です．

exsubの特徴は以下のようなものです．

(1) 改行だけを入力することができる

例えば，PIPコマンドで複数のファイルを転送する際に，

```
A>PIP
*A:=TEST.C
*A:=B:PROG.ASM
*       (改行のみ)
A>
```

などと，PIPコマンドを起動したあとで，キーボードから転送ファイル名を指定することができますが，SUBMITコマンドではXSUBを併用しても改行だけを記述することができず，PIPコマンドを終了させることができないため，必ず

```
PIP  A:=B:TEST.C
PIP  A:=B:PROG.ASM
```

として2回PIPコマンドを起動させる必要がありました．

(2) 間接的なサブミット呼出しが可能

SUBMITではそのSUBファイル中でさらにSUBMITを実行するとSUBファイルの残りの部分は無効になりました．exsubではこれを改善し，あたかもサブルーチンのように別のSUBファイルを実行させることができます．

(3) コントロールコードの入力が可能

^と大文字のA－Zを用いて，^Zなどとすればコントロールコードの入力が可能です．ただし，行先頭で^Cと記述してもリブートさせることはできません．

(4) 文字列置き換え可能

ファイル中の文字列＄0～＄9をコマンドラインで指定した文字列に置き換えることができます．例えば，TEST.SUBというファイルに対し，

```
A>@  TEST  RIE  MITARAI
```

とすると，

```
$0 → TEST
$1 → RIE
$2 → MITARAI
```

に置き換わります．この機能はSUBMITと同じです．

(5) 特殊文字の入力が可能

＄＄→＄，＄^→^という文字置き換え機能を持っています(SUBMITでは，＄＄→＄のみ)．

なお，SUBMITの補助コマンド，XSUBも@.comで利用できます．

<SUBMITの原理>

このプログラムを理解するためには，SUBMITがどのような原理で行なわれるかを知っておく必要があります．

まず，例えば，サブミットファイルとして

```
pip a:=b:test.c
ren program.c=test.c
```

という内容を持つファイル(test.sub)があると，SUBMITは＄1などファイル中の文字列の処理をしてから1行128バイトずつに展開し，最後の行から順に並べ換えたものを＄＄＄.SUBというファイルに書き込みます.

実際に実行してみると次のようになります

リスト4-1-2　SUBMITの実行

```
A>submit test

A>PIP A:=B:TEST.C

A>REN PROGRAM.C=TEST.C
A>
```

（A>PIP～A>の部分：自動的に行なわれる.）

SUBMITによって作成された＄＄＄.SUBファイルの中身は次のようになります.

リスト4-1-3　＄＄＄.SUBの構造

最初の1バイトは文字数(16進，20文字)
下線部が文字列となる.
文字列の終了は0，それ以降はゴミである.

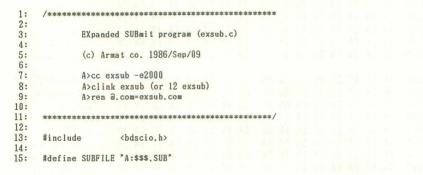
```
0100  14 52 45 4E 20 50 52 4F  47 52 41 4D 2E 43 3D 54  .REN PROGRAM.C=T
0110  45 53 54 2E 43 00 24 6F  67 72 61 6D 2E 63 3D 74  EST.C.$ogram.c=t
0120  65 73 74 2E 63 0D 0A 1A  1A 1A 1A 1A 1A 1A 1A 1A  est.c...........
0130  1A 1A 1A 1A 1A 1A 1A 1A  1A 1A 1A 1A 1A 1A 1A 1A  ................
0140  1A 1A 1A 1A 1A 1A 1A 1A  1A 1A 1A 1A 1A 1A 1A 1A  ................
0150  1A 1A 1A 1A 1A 1A 1A 1A  1A 1A 1A 1A 1A 1A 1A 1A  ................
0160  1A 1A 1A 1A 1A 1A 1A 1A  1A 1A 1A 1A 1A 1A 1A 1A  ................
0170  1A 1A 1A 1A 1A 1A 1A 1A  1A 1A 1A 1A 1A 1A 1A 1A  ................

0180  0F 50 49 50 20 41 3A 3D  42 3A 54 45 53 54 2E 43  .PIP A:=B:TEST.C
0190  00 24 54 2E 43 00 24 6F  67 72 61 6D 2E 63 3D 74  .$T.C.$ogram.c=t
01A0  65 73 74 2E 63 0D 0A 1A  1A 1A 1A 1A 1A 1A 1A 1A  est.c...........
01B0  1A 1A 1A 1A 1A 1A 1A 1A  1A 1A 1A 1A 1A 1A 1A 1A  ................
01C0  1A 1A 1A 1A 1A 1A 1A 1A  1A 1A 1A 1A 1A 1A 1A 1A  ................
01D0  1A 1A 1A 1A 1A 1A 1A 1A  1A 1A 1A 1A 1A 1A 1A 1A  ................
01E0  1A 1A 1A 1A 1A 1A 1A 1A  1A 1A 1A 1A 1A 1A 1A 1A  ................
01F0  1A 1A 1A 1A 1A 1A 1A 1A  1A 1A 1A 1A 1A 1A 1A 1A  ................
```

CP/Mのシステムはこのドライブ A の $$$.SUB ファイルの最後の1セクター（128バイト）を読み出してキーバッファーに格納すると同時に，$$$.SUB ファイルの最後の1セクターをデリートします．$$$.SUB ファイルは SUBMIT ではデフォルトドライブ（B> の時はドライブB）に作成されるのですが，CP/Mシステム側はAドライブの $$$.SUB しか検索しませんので，実際にはログインディスクがAの場合（A> のプロンプトが出ている状態）にしか利用することができません．exsubではどのような場合でも $$$.SUB はドライブAに作成されるように改善されています．

リスト4-1-3のようなフォーマットでドライブA上に $$$.SUB というファイルを作成すれば，それだけでCP/Mシステム側が自動的にプログラムの運転を行なってくれるわけです．なお，プログラム側でエラーなどの発生のため，サブミット動作を打ち切りたい時は $$$.SUB というファイルを消去します．BDS C の －x というオプションはエラー発生の際，サブミットを打ち切ってくれますが，実際には $$$.SUB を消去しているだけです．

なお，$$$.SUB ファイルは通常の使い方ではディスク上に残ることはありませんが，リスト4-1-3のようなファイルの構造を知るには，SUBMIT のバグを利用して，ドライブA以外に $$$.SUB が作成されるようにするとデバッガーなどで簡単に内容を見ることができます．

リスト4-1-4に exsub.c のソースリストを示し，詳細に解説します．

リスト4-1-4　exsub.c

```
 1:  /*************************************************
 2:
 3:           EXpanded SUBmit program (exsub.c)
 4:
 5:           (c) Armat co. 1986/Sep/09
 6:
 7:           A>cc exsub -e2000
 8:           A>clink exsub (or l2 exsub)
 9:           A>ren @.com=exsub.com
10:
11:  *************************************************/
12:
13:  #include       <bdscio.h>
14:
15:  #define SUBFILE "A:$$$.SUB"
```

```
16:  #define MAXLINE 200◄────── もっと長い行数のサブミットファイルを
17:  #define MAXCHAR 127          扱う場合にはここを変更する.
18:
19:  /*      sub file record struct  */
20:
21:  struct  subsect◄────── $$$.SUBの内部構造を表わす構造体タグ.
22:          {
23:                  char    cnt;            /* how many charactors          */
24:                  char    str[127];       /* string (null terminated)     */
25:          };
26:
27:  FILE    iobuf;                          /* input file buffer            */
28:
29:  struct  subsect secbuf[MAXLINE];        /* output buffer MAX=200 lines  */
30:  int     linecnt;                        /* how many lines               */
31:
32:  main(argc,argv)
33:          int     argc;
34:          char    *argv[];
35:  {
36:          char    filename[20];           /* filename buffer              */
37:
38:          if (argc == 1)
39:          {
40:                  printf ("Usage: @ submitfile arg1 arg2 ...");
41:                  kexit ();
42:          }
43:          strcpy (filename, argv[1]);     /* set filename                 */
44:          strcat (filename, ".SUB");
45:
46:          if (ERROR == fopen (filename, iobuf))
47:          {
48:                  printf ("Can't open %s",filename);
49:                  kexit ();
50:          }
51:          submit (argc-1,&argv[1]);       /* ex-sub main function         */
52:          fclose (iobuf);
53:          writesub ();                    /* write $$$.sub                */
54:  }
55:
56:
57:  submit(ac,av)
58:          int     ac;
59:          char    *av[];
60:  {
61:          char    buf[150];               /* fgets line buffer            */
62:          char    *pnt;                   /* submit file buffer pointer   */
63:
64:          linecnt = 0;
65:          while (fgets(buf,iobuf))
66:          {
67:                  dellf (buf);
68:                  pnt = secbuf[linecnt].str;
69:                  *pnt = 0;
70:                  argexp (ac,av,buf,pnt);         /* set $1,$2 ...        */
71:
72:                  if ((secbuf[linecnt].cnt = strlen(pnt))>MAXCHAR)
73:                  {
74:                          printf ("Line overflow at line %d",linecnt+1);
75:                          kexit ();
76:                  }
```

```
 77:                     linecnt++;
 78:             }
 79:     }
 80:
 81:
 82:     dellf(buf)
 83:             char    *buf;
 84:     {
 85:             while (*buf != '¥n')
 86:             {
 87:                     if (*buf == 0)  break;
 88:                     buf++;
 89:             }
 90:             *buf = 0;
 91:     }
 92:
 93:
 94:     argexp(ac,av,buf,pnt)
 95:             int     ac;
 96:             char    *av[];
 97:             char    *buf,*pnt;
 98:     {
 99:             int     chars;
100:             int     i;
101:             int     strpnt;
102:             while (1)
103:             {
104:                     strpnt = index (buf,"$");←―――― $があるかどうかのチェック.
105:                     if (strpnt == ERROR)
106:                     {
107:                             strcopy (pnt,strlen(buf),buf);←―――― なかった場合にはその
108:                             break;                                ままコピー.
109:                     }
110:                     pnt = strcopy (pnt,strpnt,buf);
111:                     buf += strpnt+1;
112:                     if (isdigit(*buf))
113:                     {
114:                             i = *buf-0x30;
115:                             if (i<ac)  pnt = strcopy (pnt,strlen(av[i]),av[i]);
116:                             buf++;
117:                     }
118:                     else if (*buf == '$')
119:                     {
120:                             *pnt++ = '$';
121:                             buf++;
122:                     }
123:                     else if (*buf == '^')
124:                     {
125:                             *pnt++ = '^';
126:                             buf++;
127:                     }
128:             }
129:     }
130:
131:
132:     strcopy(pnt,chars,buf)                  /* string copy with control charactors*/
133:             char    *pnt,*buf;
134:             int     chars;
135:     {
136:             char    c;
137:             while (chars--)
```

```
138:            {
139:                    if (*buf == '^')        /* control charactor    */
140:                    {
141:                            buf++;
142:                            c = toupper(*buf++);
143:                            if (c >= 0x40 || c< 0x60)       *pnt++ = c-0x40;
144:                            if (chars--)    break;
145:                    }
146:                    else    *pnt++ = *buf++;
147:            }
148:            *pnt = 0;
149:            return (pnt);
150:    }
151:
152:
153:    writesub()                              /* write submit file    */
154:    {
155:            struct  subsect buf;
156:            int     i;
157:            int     fd;
158:            if (ERROR !=  ( fd = open (SUBFILE,1))) seek (fd,0,2);
159:            else if ( ERROR == (fd = creat (SUBFILE)))
160:            {
161:                    printf ("Can't creat ¥"$$$.SUB¥"");
162:                    kexit ();
163:            }
164:
165:            for (i=linecnt-1; i>=0; i--)
166:            {
167:                    if (1 != write (fd,secbuf[i],1))
168:                    {
169:                            printf ("Disk full...");
170:                            kexit ();
171:                    }
172:            }
173:            close (fd);
174:    }
175:
176:
177:    kexit()
178:    {
179:            unlink (SUBFILE);←――――unlinkはファイルを消去する関数.
180:            exit ();
181:    }
182:
```

[15〜17行]

プログラム中で用いる定数およびファイル名を設定しています．ここでMAXLINEというのは変換可能なSUBファイルの行数を示しています．1行につき，128バイト消費しますので200ラインで25Kバイト必要です．メモリが許す範囲で値を大きく設定すればもっと多くの行数のファイルを処理できるようになります．

MAXCHARはサブミットで可能な1行当りの文字数です．変更はできませ

ん．

［21〜25行］

subsectという構造体タグを定義しています．$$$.SUBファイルの構造と対応させ，扱いやすくしているわけです．

［29行］

subsect構造をした構造体配列secbufを宣言しています．ここには実際にディスクに書き込む $$$.SUBのイメージがそのまま作成されます．

［38〜42行］

ここはプログラムの利用方法を示すUsage：を表示する部分です．@だけで改行すると，このメッセージが表示されます．

［43〜50行］

ここで，目的のSUBファイルの名前を文字列filenameに格納し，そのファイルをfopenでオープンします．iobufは27行めで宣言されていますが，バッファードファイルIOのためのバッファーのアドレスです．

もし，目的のファイルがオープンできなければ，エラーですからそこで中断します．

［51行］

このプログラムの実際の処理をおこなう，submitという関数を呼び出します．

［52，53行］

オープンしたファイルをクローズし，メモリ上のサブミットファイルのイメージを実際にディスクに書き込みます．

[57～79行]

実際の処理を行なう submit関数です．

fgets により1行ずつファイルを読み出し，＄1，＄2などの処理を行ないながら，secbuf に書き込んでいきます．この時，一旦文字列 buf にその内容を書き込み，その長さがMAXCHAR(127)より短いことを確認しながら行なっていきます．

[82～91行]

関数 dellf です．fgets で取り込んだ1行は最後に ¥n コード（0AH）が付いています．これを $$$.SUB ファイルの中に書き込んではいけないので，削除する関数です．

[94～129行]

＄1，＄2などの文字列にコマンドラインからの文字列を展開する関数argexpです．引数 ac，av は main 関数への引数とほとんど同じですが，51行めで判るように，あらかじめ不要な部分（argv [0]）は除いてあります．pnt が書き込みのポインタ，buf が読み込みのポインタで，buf からの文字を読みながら，＄1，＄2などの処理を行ない，pnt に書き込みます．

[132～150行]

通常の strcpy とほとんど同じ機能を持つ関数 strcopy です．cnt という引数により，文字列を指定の長さまでコピーします．また，^とA，^とZなど文字^と一緒のアルファベットを実際のコントロールコードに直します．

[153～174行]

関数 writesub は submit 関数によりメモリ secbuf 上に展開されている入力ファイルの内容をドライブAに $$$.SUB というファイル名で書き込みます．この時，最後から書き込む点に注意してください．

なお，158，159行で，$$$.SUBというファイルがオープンできれば（存在すれば），そのファイルの最後から書き込むように

```
seek (fd,0,2) ;
```

とし，リードライトポインタをファイルの最後に設定します．また，もしオープンできなければ（存在しなければ），ファイルを新たにcreatしておきます．

これで，サブミット作業中にこのプログラムが呼ばれた場合，新たな作業内容が付け足され，あたかもサブルーチンのように実行されるわけです．

［177〜181行］

エラーなどが発生した時には$$$.SUBをデリートしてから終了します．

exsub.cはリストの7〜9行にある要領でコンパイル・リンクし，最後に得られたCOMファイルを@.comという名前にリネームしてください．

4-2 拡張サブミット補助コマンド/.com

@.comはSUBMITの機能をそのまま拡張したものですから，複雑な処理を行なわせることはできません．そこで，サブミット動作を補助するコマンドがあると便利です．

exsubをサポートするコマンド/（スラッシュ）の機能を説明します．

/.comはプログラム名/の後にスペースを空けてサブコマンド名を記述することでサブミット実行時に色々な機能を利用することができるようになります．

```
/ if file 実行文字列
```

fileが存在すれば実行文字列を実行します.

例えば，exsub.bakというファイルの有無をチェックし，exsub.cをコンパイルするかどうかを制御するなどといった事が可能です.

```
/ nif file 実行文字列
```

fileが存在しなければ，実行文字列を実行します.

```
/ eq 文字列1 文字列2 実行文字列
```

文字列1と文字列2が一致すれば実行文字列を実行します.

$1，$2などでコマンドラインから文字列を設定すれば，subファイルの実行をコントロールすることができます.

```
/ neq 文字列1 文字列2 実行文字列
```

文字列1と文字列2が一致しなければ実行文字列を実行します.

```
/ skip 行数
```

無条件で行数分実行をキャンセルします.

このコマンドは/nifなどと組み合せて用いられ，行数はサブミットファイル内でこの命令以降に記述されている行数が最大です.

〈例〉　/ nif file.bak / skip 3

/ stop

サブミットファイルの実行を終了します．/ skipと同様，/ ifなどと組み合せて用いられます．

通常プログラム開発中には，エディターを用いてプログラムを書き直し，さらにそのプログラムをコンパイルする必要があるわけですが，このとき，バックアップファイルとして，拡張子が，bakというファイルが作成されますから，次のようなサブミットファイルを作ることができます．

リスト4-2-1　/.comのサンプル

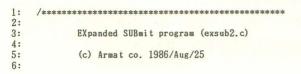
```
/ if exsub.bak @ cc exsub -e2000
/ if exsub2.bak @ cc exsub2
```

- もし exsub. bakが存在すれば，@ cc exsub －e2000を実行する．
- もし exsub2. bakが存在すれば，@ cc exsub2を実行する．
- （注：このファイルを実行する時はリスト4－1－1のcc. subが必要．）

これで，bakファイルがある場合（ファイルを更新した場合）には，コンパイルが行なわれ，無い場合にはコンパイルが行なわれず，無駄を省くことができるわけです．なお，この /. comは@.comではなく，通常のSUBMITと組み合せて使うこともできます．

それでは，/.comのソースプログラム，exsub 2.cについて説明します．

リスト4-2-2　exsub 2.c

```
1:    /***************************************************
2:
3:            EXpanded SUBmit program (exsub2.c)
4:
5:            (c) Armat co. 1986/Aug/25
6:
```

```
 7:         A>cc exsub2
 8:         A>clink exsub2 (or 12 exsub2 )
 9:         A>ren /.com=exsub2.com
10:
11: ************************************************/
12:
13: #include        <bdscio.h>
14:
15: /*      sub file record struct  */
16:
17: struct  subsect
18:         {
19:                 char    cnt;            /* how many charactors          */
20:                 char    str[127];       /* string (null terminated)     */
21:         };
22:
23: #define SUBCOM  1       /* SUBCOM is arg number in command line */
24: #define MAXCHAR 127
25: #define YES     1
26: #define NO      0
27:
28: main(argc,argv)
29:         int     argc;
30:         char    *argv[];
31: {
32:
33:         if (argc == 1)
34:         {
35:                 printf ("Usage: / if  testfile command arg...¥n");
36:                 printf ("       / nif testfile command arg...¥n");
37:                 printf ("       / eq  string1 string2 command arg...¥n");
38:                 printf ("       / neq string1 string2 command arg...¥n");
39:                 printf ("       / skip lines¥n");
40:                 printf ("       / stop¥n");
41:                 kexit ();
42:         }
43:
44:         if      (!strcmp (argv[SUBCOM],"IF"))   iffunc (argc,argv,YES);
45:         else if (!strcmp (argv[SUBCOM],"NIF"))  iffunc (argc,argv,NO);
46:         else if (!strcmp (argv[SUBCOM],"EQ"))   equal (argc,argv,YES);
47:         else if (!strcmp (argv[SUBCOM],"NEQ"))  equal (argc,argv,NO);
48:         else if (!strcmp (argv[SUBCOM],"SKIP")) skip (argc,argv);
49:         else if (!strcmp (argv[SUBCOM],"STOP")) kexit ();
50:         else
51:         {
52:                 printf ("%s..?¥n",argv[SUBCOM]);
53:                 kexit ();
54:         }
55: }
56:
57: iffunc(argc,argv,yorn)
58:         int     argc;
59:         char    *argv[];
60:         char    yorn;
61: {
62:         int     fd;
63:
64:         if (argc < SUBCOM+2)    argerr ();
```

```
 65:         fd = open (argv[SUBCOM+1],0);
 66:         if ( (ERROR == fd && yorn == NO) ||
 67:              (ERROR != fd && yorn == YES) )
 68:                 nextsub (argc-(SUBCOM+2),&argv[SUBCOM+2]);
 69:         if (ERROR != fd)        close (fd);
 70: }
 71:
 72:
 73: nextsub(ct,args)                /* write next command to submit file    */
 74:         int     ct;
 75:         char    *args[];
 76: {
 77:         struct  subsect sec;
 78:         char    buf[200];
 79:         int     i;
 80:         int     fd;
 81:
 82:         strcpy (buf,args[0]);
 83:         for (i=1; i<ct; i++)
 84:         {
 85:                 strcat (buf," ");
 86:                 strcat (buf,args[i]);
 87:                 if (strlen (buf) > MAXCHAR)
 88:                 {
 89:                         printf ("Line overflow ..");
 90:                         kexit ();
 91:                 }
 92:         }
 93:         sec.cnt = strlen (buf);
 94:         strcpy (sec.str,buf);
 95:                                         /* write disk file      */
 96:         if (ERROR == (fd = open ("A:$$$.SUB",1)))
 97:         {
 98:                 printf ("Not in submit.");
 99:                 kexit ();
100:         }
101:         seek (fd,0,2);<────────ファイルにアペンドするために
102:         if (1 != write (fd,sec,1))  リードライトポインタをファイル
103:         {                           の最後にセット.
104:                 printf ("Disk full ..");
105:                 kexit ();
106:         }
107:         close (fd);
108: }
109:
110:
111: equal(argc,argv,yorn)
112:         int     argc;
113:         char    *argv[];
114:         char    yorn;
115: {
116:         if (argc < SUBCOM+3)    argerr ();
117:         if (yorn == (!strcmp(argv[SUBCOM+1],argv[SUBCOM+2])))
118:                 nextsub (argc-(SUBCOM+3),&argv[SUBCOM+3]);
119: }
120:
121:
122: skip(argc,argv)
```

```
123:            int      argc;
124:            char     *argv[];
125:    {
126:            int      sline;           /* how many skip lines                          */
127:            if (argc < SUBCOM+1)      argerr ();
128:            sscanf (argv[SUBCOM+1],"%d",&sline);
129:            delline (sline);
130:    }
131:
132:
133:    delline(sline)              /* delete submit lines                            */
134:            int      sline;
135:    {
136:            struct   subsect buf;
137:            int      i;
138:            int      fd;
139:
140:            if (ERROR == (fd = open ("A:$$$.sub",1)))        exit ();
141:
142:            buf.str[0] = 0;
143:            buf.cnt = 0;
144:            for (i=1; i<=sline; i++)
145:            {
146:                    if (ERROR == seek (fd,-i,2))
147:                    {
148:                            printf ("Can't delete lines");
149:                            kexit ();
150:                    }
151:                    write (fd,buf,1);
152:            }
153:            close (fd);
154:    }
155:
156:
157:    argerr()
158:    {
159:            printf ("Strange line..");
160:            kexit ();
161:    }
162:
163:
164:    kexit()
165:    {
166:            unlink ("A:$$$.SUB");
167:            exit ();
168:    }
169:
```

［17～26行］

ここでは exsub.c と同じ構造体のテンプレートを宣言しています.

SUBCOM は if, eq などのサブコマンドがコマンドライン中でどの位置にあるかを示す数値で，第1番めの引数ですから1となっています.

[33〜42行]

/だけをタイプし，改行すると操作方法が表示されます．

[44〜54行]

サブコマンドの文字列によって，その処理をする関数を呼び出します．新しい機能を追加したければ，この部分に付け足せば良いでしょう．勿論，該当するコマンドがなければエラーですから，エラーメッセージを表示して終了します．

[57〜70行]

サブコマンドifとnifを処理する関数です．yornという引数は“yes or no”のことで，if(あれば)とnif（なければ)が両方同一の関数で処理できるように設けられています．この関数中で使われているopenは実際のファイル読み書きのためではなく，ファイルがあるかないかを確認する目的で行なっています．ファイルの有無を確認するのは，他にwildexpパッケージなど用いてもできますが，こちらのほうが簡単です．

条件が合えば，関数nextsubに必要な文字列の数とポインタ配列の先頭アドレスを送り，その内容を$$$.SUBに書き込みます．

[73〜108行]

文字列へのポインタ配列の先頭アドレスをargsとし，ct個の文字列を$$$.SUBファイルにアペンドする関数です．引数のフォーマットはコマンドラインからの引数を受け取る方法と全く同じです．

一旦bufに文字列を設定し，それから実際に書き込むsecにコピーしています．これは展開する文字列をじかにsecに格納してしまうと文字列の長さが127文字を越えた場合に暴走する可能性があるためですが，実際にはコマンドラインそのものに127文字以上記述することができないため，不必要とも言えます．しかし将来プログラムを変更する際など，不要なトラブルに巻き

込まれないですみます.

[111〜119行]

文字列の比較をして，もし条件が合えばコマンドラインの文字列を $$$.SUBに書き込むeqおよびneqの処理ルーチンです.

[122〜130行]

skipの処理をする関数です. sscanfで文字列を数値に変換し，デリートする行数を設定して関数dellineを呼び出します.

[133〜154行]

サブミットで実行される処理を無効にする関数dellineです. 無効にするためには $$$.SUBファイルの最後から行数 *128バイト分消去してしまえば良いのですが，BDS Cの標準関数ではファイルの一部を消去する関数がない（CP/MのBDOSコールそのものに無い）ため，コマンド入力が無効になるよう文字数0となるようにファイルの内容を書き換え，実現しています(ディスクのディレクトリをじかに変更すれば可能です). そのため，skipが実行された場合にはskipする行数分，入力無しで実行が行なわれます.

[157〜168行]

入力にエラーがあった場合に実行する関数argerr，サブミット動作を中止する関数kexitです.

なお，/の条件判断は実行された時点で行なわれます. @コマンド起動時に.bakファイルが存在していても/コマンド実行時までに消去されたりリネームされていれば，.bakファイルは存在しないと判断されます.

最後に，prg1.c，prg2.cという2つのBDS Cソースファイルとprgasm.csmというアセンブラソースファイルがある場合にそれをコンパイルする

SUBファイルとその実行例を紹介します.

リスト4-2-3　@と/コマンドの使用例

```
A>type prg.sub

/ if prg1.bak cc prg1 -x ←————prg1.bakが存在すれば, prg1.cをコンパイル.
/ if prg2.bak cc prg2 -x ←————prg2.bakが存在すれば, prg2.cをコンパイル.
/ if prgasm.bak @ zcmr prgasm ←——prgasm.bakが存在すれば,
                                  さらにzcmrというサブミット処理を行なう.
era *.bak
clink prg1 prg2 prgasm

A>type zcmr.sub

zcasm $1
mrasm $1.aaz    ;(drive b)->.bbz
era $1.asz
load $1                           Z80ニーモニックで書かれたソースファイル
era $1.hex                        (ここではprgasm.csm)をアセンブルし,
era $1.crl                        BDS CのCRLリンク形式に変換する一連の
ren $1.crl=$1.com                 処理.
; $1.crl is ready


A>dir prg*.c*
A: PRG1     C    : PRG2     C    : PRGASM   CSM ←———ソースファイル

A>dir prg*.bak
A: PRG1     BAK  : PRG2     BAK  : PRGASM   BAK ←—すべてバックアップ
                                                  ファイルが存在して
                                                  いる.
A>type prg1.c

#include <bdscio.h>

main()
{                                                 prg1.cの内容.
        char    i;
        for (i=0; i<8; i++)     prg2 (i);
}
```

```
A>type prg2.c

#include <bdscio.h>

prg2(i)
        char    i;
{
        printf ("original=%d,decode=%d¥n",i,prgasm(i));
}
```

prg2.cの内容.

```
A>type prgasm.csm

function prgasm

        pop     hl      ; return address
        pop     de      ; argument
        push    de
        push    hl

        ld      hl,table
        add     hl,de
        ld      l,(hl)
        ld      h,0
        ret

; decode table
table:  db      1,2,4,8,16,32,64,128

endfunc
```

prgasm.csmの内容0～7の値をデコードし,
0 → 1
1 → 2
2 → 4
3 → 8
4 → 16
5 → 32
6 → 64
7 → 128
に変換する関数.

```
A>@ prg

A>/ if prg1.bak cc prg1 -x

A>CC PRG1 -X
BD Software C Compiler v1.50a (part I)
  35K elbowroom
BD Software C Compiler v1.50 (part II)
  32K to spare
```

A>@ prg ← サブミットの起動.
以下は自動的に行なわれる.

prg1.bakが存在するので,
prg1.cがコンパイルされた.

```
A>/ if prg2.bak cc prg2 -x

A>CC PRG2 -X
BD Software C Compiler v1.50a (part I)
  35K elbowroom
BD Software C Compiler v1.50 (part II)
  32K to spare
```

prg2.bakが存在するのでprg2.cがコンパイルされた.(なければコンパイルされない)

```
A>/ if prgasm.bak @ zcmr prgasm

A>@ ZCMR PRGASM
```

prgasm.bakが存在するのでzcmrというサブミットファイルが起動される.

```
A>zcasm PRGASM
Armat Z80-CRL preprocessor v1.0
Pass 1: End
Pass 2:
 -Processing the PRGASM function..
  PRGASM.ASZ is ready to be assembled.

A>mrasm PRGASM.aaz    ;(drive b)->.bbz
Armat Z80 Assembler - VER 1.3
   (c) Armat co. 1985,1986 All Rights Reserved.

No Fatal error(s)

A>era PRGASM.asz
A>load PRGASM

FIRST ADDRESS 0100
LAST  ADDRESS 031F
BYTES READ    002B
RECORDS WRITTEN 05


A>era PRGASM.hex
A>era PRGASM.crl
NO FILE
A>ren PRGASM.crl=PRGASM.com
A>; PRGASM.crl is ready
```

zcmr内の処理.

```
A>era *.bak

A>clink prg1 prg2 prgasm
BD Software C Linker   v1.50
Linkage complete
  43K left over
```

zcmrの処理が終了すると元のprg.subの処理に戻る.

〈サブミット動作終了〉

```
A>prg1
original=0,decode=1
original=1,decode=2
original=2,decode=4
original=3,decode=8
original=4,decode=16
original=5,decode=32
original=6,decode=64
original=7,decode=128

A>
```

プログラムの実行

第 5 章
日本語処理

BDS C，α-Cでは基本的に日本語の処理ができるようにはなっていません．本来，これらは米国で開発されたものですから，文字としては英数字だけが想定されており，文字列処理用の標準関数などもそのままでは使うことができない場合がほとんどです．

しかし，C言語の基本的な仕様は（日本語，英語などといった）言語に左右されてはおらず，新しい関数を作成すれば本格的な日本語処理を行なうことが可能です．本章では日本語を使う際のポイントを解説し，ならびに実用的な日本語処理関数パッケージを紹介します．

5-1 BDS C, α-Cの変更

BDS C, α-Cでは基本的な文字として英数字だけが考えられていますので，カナ，漢字などはソースリストに記述してもコンパイラにかけた時点でアスキーコードのMSBが落とされてしまい，英数字に変化してしまいます．

リスト5-1-1 BDS Cでカナを用いた場合

```
A>type kana.c

main()
{
        printf ("ｶﾅﾊ ｵﾘｼﾞﾅﾙ ﾉ BDS C ﾃﾞﾊ ﾂｶｴﾏｾﾝ.");
}

A>cc kana -p
BD Software C Compiler v1.50a (part I)

   1: main()
   2: {
   3:   printf ("6EJ 5X<^EY I BDS C C^J B640>].");
   4: }

  36K elbowroom
BD Software C Compiler v1.50 (part II)
  32K to spare

A>clink kana
BD Software C Linker   v1.50
Linkage complete
  43K left over

A>kana
6EJ 5X<^EY I BDS C C^J B640>].

A>
```

MSB（最上位ビット）が落とされるため，このようになってしまう．

表示はもちろんこのようになる．

リスト5-1-1の状況では，全く日本語の処理などできなくなります．し

かし，strcpy，strlenなどの標準的な文字列処理関数はこの制限を持っているわけではなく，終端文字の0さえ確実にセットしてあればMSBは影響を受けません．そこで，コンパイル時にカナ，漢字のMSBがキャンセルされないようにすれば，何とか日本語処理ができるようになります．

この方法として次のようなものが考えられます．

(1)　コンパイラ自体を変更し，MSBがキャンセルされないように改造する．

(2)　ソースファイル用のプリプロセッサを作成し，カナ，漢字をコンパイラで正しく認識されるような形式に変更する．

(1)は簡単なパッチ処理で済みますが，本質的な方法ではないため，多くの制限が残ります．(2)は処理は完全ですが，別のプリプロセッサにソースファイルを通さねばならないため，コンパイル時の手間がかなりかかります．

実用的には両者を併用するのが便利でしょう．なお，本項で扱う漢字は，シフトJISコード方式のものです．

BDS Cはコンパイラ本体はcc.comおよび，cc2.comという2つのプログラムから成り立っていますが，このうち，文字列の解析などを行なっているのはプリプロセッサ部であるcc.comです．cc.comの次の番地の内容を7FHからFFHに変更することにより，MSBのキャンセルを防ぐことができます．

BDS C	v1.50a	27C7H番地
α-C	v1.51	27F1H番地

操作例5-1-2　cc.comの変更

```
A>ddt cc.com
DDT VERS 2.2
NEXT  PC
3A00 0100
```

```
-s27c7 ←DDTなどによって27C7H番地の内容を
         変更する（α-Cの場合は27F1H番地）.
27C7 7F ff
27C8 22 .
-^C

A>save 57 cc.com ←SAVEコマンドにより書き換えた内容を記録する.

A>cc kana -p ←変更を加えたBDS Cでテスト.
BD Software C Compiler v1.50a (part I)

   1: main()
   2: {
   3:   printf ("ｶﾅﾊ ｵﾘｼﾞﾅﾙ ﾉ BDS C ﾃﾞﾊ ﾂｶｴﾏｾﾝ.");
   4: }

  36K elbowroom
BD Software C Compiler v1.50 (part II)
  32K to spare

A>clink kana
BD Software C Linker   v1.50
Linkage complete
  43K left over

A>kana

ｶﾅﾊ ｵﾘｼﾞﾅﾙ ﾉ BDS C ﾃﾞﾊ ﾂｶｴﾏｾﾝ. ←正しく表示される.

A>
```

この方法は非常に手軽ですが，残念ながら，次のような欠点を持っています．

(1)　マクロ定義された文字列については有効でない．

プログラム内部で

```
printf （“漢字テスト”）；
```

などと記述する場合には問題ありませんが，次のような場合は正しくコンパイルされません．

リスト5-1-3　マクロ定義でうまくいかない例

```
A>type define.c

#define MSG      "ｶﾅ ﾊ ﾏｸﾛﾃｲｷﾞ ﾃﾞﾊ ﾂｶｴﾏｾﾝ｡"  ←文字列のマクロ定義.

main()
{
        printf (MSG);
}

A>cc define -p
BD Software C Compiler v1.50a (part I)

   1:
   3: main()
   4: {
   5:   printf ("6);
   6: }

   5: String too long (or missing quote)  ←この場合はコンパイルそのものができなかった.

A>
```

余り変更しないメッセージはじかにプログラム内部に置く場合が多いのですが，変更が考えられるメッセージなどはあらかじめ#defineを用いてマクロ定義とする方がはるかに便利ですし，間違いも少なくなります．これはかなり大きな制約であると言えます．

(2)　特別な漢字を用いた場合に文字化けが発生する．

C言語では，文字￥（JISコード．アスキーコードでは/）を文字，文字列中の特別な記号として用います．所が，この記号は5CHというコードを持っていますから，シフトJIS方式の2バイトコードで漢字を表示すると，2バイ

トめがこのコードと一致することがあります．この場合，文字列の内容を誤って解釈してしまいますから，完全に文字が変化することになります．さらに具合が悪いことに文字列の最後にこの文字がある場合には文字化けでは済まず，コンパイルそのものができなくなります．

リスト5-1-4　文字化けする場合

```
A>cc kanjerr -p
BD Software C Compiler v1.50a (part I)

   1: main()
   2: {
   3:   printf ("すべての漢字を正しく表現できません");
   4: }
                                  └─コンパイルでは正しいように
                                    見えるが……
  36K elbowroom
BD Software C Compiler v1.50 (part II)
  32K to spare

A>clink kanjerr
BD Software C Linker   v1.50
Linkage complete
  43K left over

A>kanjerr

すべての漢字を正しく圃りできません←――文字化けした文字列．漢字コード
                                      の2バイトめが5CHの文字はこのよ
                                      うになってしまう．
A>
```

リスト5-1-5　コンパイルができない場合

```
A>cc f:kanjerr2 -p
BD Software C Compiler v1.50a (part I)

   1: main()              どう見ても正しいように見えるが，
   2: {                   "がないと判断される．
   3:   printf ("人数表"); これは「表」の２バイトめが5CH(￥)
   4: }                   で，￥"と解釈されたため．
```

```
3: String too long (or missing quote)

A>
```

パッチによる方法は非常に簡単で効果も大きいのですが，そこに上記のような問題が残されている事にも注意しなければなりません．

5-2　プリプロセッサを用いる方法

あらかじめソースプログラムそのものを変更してしまおうというのがプリプロセッサを用いる方法ですが，C言語ではもともと8ビットフルに用いた文字列を使ってはならないわけではなく，ある特定の条件のもとでは使用することが可能です．

"¥377¥376"

という文字列は特殊文字¥を用いて，0FFH，0FEHのコードを文字列に設定する例です．¥マークのあと8進3桁で数値を指定することにより0から0FFHまですべてのコードを文字列の中に埋め込めます．この方法を用いて，

"表現"──→　"¥225¥134¥214¥273"

という形に変換してしまうわけです．全く何を表現しているのか判らなくなりますが，すべての制約はなくなります．

それでは，この処理を行なうプリプロセッサktocをご紹介します．

リスト5-2-1　ktoc.c

```
1:   /*****************************************************
2:
3:             日本語コード変換ユーティリティ
4:                     (ktoc.c)
5:
6:             Author: Tsu.Mitarai      12/Dec/1986
7:
8:                     A>cc ktoc -e2100
```

```
 9:                 A>clink ktoc
10:
11: ****************************************************/
12: #define DEBUG   0
13:
14: #include        <bdscio.h>
15:
16: char    lbuffer[512];   /* get line buffer     */
17: FILE    fpin;           /* input file buffer   */
18: FILE    fpout;          /* output file buffer  */
19:
20: int     lcnt;           /* 行カウンター        */
21: char    kanjif;         /* 漢字あり:1,カナのみ:0*/
22:
23:
24: main(argc, argv)
25: int     argc;
26: char    *argv[];
27: {
28:         char    infile[20];
29:         char    outfile[20];
30:
31:         if ( argc < 2 )
32:         {
33:                 printf ("書式: ktoc filename [-nk]¥n");
34:                 printf ("  -nk: 漢字を使用しない場合");
35:                 exit ();
36:         }
37:         strcpy (infile,argv[1]);
38:         strcpy (outfile,argv[1]);
39:         strcat (infile,".C");
40:         strcat (outfile,".CK");
41:
42:         if (strcmp (argv[2],"-NK"))     kanjif = TRUE;
43:         else                            kanjif = FALSE;
44:
45:         if (ERROR == fopen (infile,fpin))
46:         {
47:                 printf ("<%s>がオープンできません",infile);
48:                 exit ();
49:         }
50:         if (ERROR == fcreat (outfile,fpout))
51:         {
52:                 printf ("<%s>を作成することができません",outfile);
53:                 exit ();
54:         }
55:         ktoc ();                /* メインルーチン */
56:         fputc (CPMEOF);
57:
58:         if (ERROR == fclose (fpout))
59:                 printf ("<%s>を正しくクローズできません",outfile);
60:         else    printf ("<%s>を作成しました",outfile);
61: }
62:
63:
64: ktoc()
65: {
66:         char    c1,c2;
67:         char    *cpnt;
68:         int     comcnt;         /* コメントのネストカウンター */
```

```
 69:
 70:          lcnt = 1;
 71:          comcnt = 0;
 72:          while ( fgets (lbuffer, fpin) )
 73:          {
 74: #if DEBUG
 75:                  printf ("(%d)%s",comcnt,lbuffer);
 76: #endif
 77:                  cpnt = lbuffer;
 78:                  c1 = 0;
 79:                  while (c2 = *cpnt)
 80:                  {
 81:                          fputc (c2);
 82:                          cpnt++;
 83:                          if (c1=='/' && c2=='*')         comcnt++;
 84:                          else if (c1 == '*' && c2 == '/')
 85:                          {
 86:                                  if (--comcnt < 0)
 87:                                  {
 88:                                          printf ("%d: %s",lcnt,lbuffer);
 89:                                          printf ("コメントが正しくありません");
 90:                                          exit ();
 91:                                  }
 92:                          }
 93:                          if (!comcnt)
 94:                          {
 95:                                  if (c2 == '¥'') cpnt=skipchar (cpnt);
 96:                                  if (c2 == '"')  cpnt=setstr (cpnt);
 97:                          }
 98:                          c1=c2;
 99:                  }
100:                  lcnt++;
101:          }
102:          if (comcnt !=0)
103:          {
104:                  printf ("コメントが閉じていません");
105:                  exit ();
106:          }
107: }
108:
109:
110: fputc(c)
111:          char    c;
112: {
113:          if (c=='¥n')    putc ('¥r',fpout);
114:
115:          if ( ERROR == putc (c,fpout))
116:          {
117:                  printf ("ディスクが一杯です");
118:                  exit ();
119:          }
120: #if DEBUG
121:          putchar (c);
122: #endif
123: }
124:
125:
126: skipchar(pnt)
127:          char    *pnt;
128: {
```

```
129:            while (*pnt != '¥'')
130:            {
131:                    fputc (*pnt);
132:                    if (*pnt == '¥¥')       fputc (*++pnt);
133:                    else if (*pnt == '¥0')
134:                    {
135:                            printf ("%d: %s",lcnt,lbuffer);
136:                            printf ("文字が正しく記述されていません");
137:                            exit ();
138:                    }
139:                    pnt++;
140:            }
141:            fputc (*pnt++);
142:            return (pnt);
143:    }
144:
145:
146:    setstr(pnt)
147:            char    *pnt;
148:    {
149:            char    kflag;          /* 漢字 1バイトめのフラグ */
150:
151:            kflag = FALSE;
152:            while (1)
153:            {
154:                    if (kflag)
155:                    {
156:                            if (*pnt=='¥¥' || *pnt>=0x80)
157:                            {
158:    #if DEBUG
159:                                    printf ("(2nd)¥¥%03o",*pnt);
160:    #endif
161:                                    fprintf (fpout,"¥¥%03o",*pnt++);
162:                            }
163:                            else    fputc (*pnt++);
164:                            kflag = FALSE;
165:                    }
166:                    else if (*pnt=='¥¥')
167:                    {
168:                            fputc (*pnt++);
169:                            fputc (*pnt++);
170:                    }
171:                    else if (*pnt>=0x80)
172:                    {
173:    #if DEBUG
174:                            printf ("(1st)¥¥%03o",*pnt);
175:    #endif
176:                            fprintf (fpout,"¥¥%03o",*pnt);
177:                            if ( kanjif             &&
178:                                ((*pnt>=0x80 && *pnt<0xa0) ||
179:                                 (*pnt>=0xe0 && *pnt<0xfd))  )
180:                                    kflag = TRUE;
181:                            pnt++;
182:                    }
183:                    else if (*pnt == '"')
184:                    {
185:                            fputc (*pnt++);
186:                            return (pnt);
187:                    }
188:                    else if (*pnt == '¥0')
```

```
189:                    {
190:                            printf ("%d: %s",lcnt,lbuffer);
191:                            printf ("文字列が長過ぎるか¥"マークが落ちています");
192:                            exit ();
193:                    }
194:                    else    fputc (*pnt++);
195:            }
196:    }
197:
```

〈変数および関数表〉

lbuffer…………ファイルからの行入力バッファー
fpin…………………入力用FILE構造体
fpout………………出力用FILE構造体
lcnt…………………入力ファイル行番号
kanjif……………漢字処理をするかどうかのフラグ

main()……………入出力用ファイル名の設定，ファイルのオープクローズ処理
ktoc()……………プログラムの本体部分
fputc(c)…………出力ファイルへの1文字出力用の関数
skipchar()……' ' で囲まれた文字定数の処理
setstr()…………" " で囲まれた文字列の処理関数
（漢字→8進数変換はここで行なっている）

このktocはシフトJIS漢字コード方式の場合，その第1バイトめが80H～9FHおよびE0H～FCHにあることを利用しています．

プログラム内部の文字列をすべてチェックし，文字列の中に80H以上のキャラクタがあればそれを¥＋8進3桁の形式に変換して出力します．ただし，漢字の2バイトめの場合，そのコードが¥そのもののコード（5CH）と同じだとコンパイル時に誤って変換されてしまいますので，5CH，および80H以上の場合のみ¥＋8進3桁の形式に直します．それ以外は文字キャラクタとなりますが，これはBDS Cでは文字列として最大255文字までしか記述できず，¥＋8進3桁で記述した場合，見掛け上文字列が長くなり，コンパイル不能になることがあるからで，少しでも短くなるようにしてあります．

なお，漢字を用いず，カタカナだけを利用する場合は，逆に80H～9FH，E0H～FCHに設定されているグラフィックキャラクタを利用すると問題を生ずるため，ktocプリプロセッサにかける時にオプションを指定できるようになっています．

ktocの使い方は次のとおりです．

```
書式：ktoc filename [−nk]
```

filenameに拡張子”・c”はいりません．また出力ファイル名はfilenameに拡張子”.ck”を付けたものとなります．またオプションの−nkは漢字を使わない場合に指定します（no kanji）

正しく変換されると，”<filename.ck> を作成しました”というコメントが出て終了しますので，あとは

```
cc filename.ck
clink filename
```

という通常の方法でコンパイルすることができます．注意することはコンパイル時に必ずファイル名に ”.ck”という拡張子を忘れないことで，これを忘れると元のファイルがコンパイルされることになります．

なお，このktocはエラーメッセージなどがすべて漢字で記述されているため，5-1項で示された変更を行なったBDS Cあるいはα-Cでなければコンパイルできません（メッセージを英字にしておけばコンパイルできます）．しかし，ktocで変換されたファイルはcc.com変更前のものでもコンパイルできますので，このプログラムを入力された方は，ktoc.cそのものをktocにより変換し，正しく動作するか確かめると良いでしょう．なお，ktocにかけるプログラムでは文字列を1行以内で記述してください．文字列中で改行するとエラーとなります．また，文字（’ ’で囲んだもの）中の漢字などは変換されません．

次に操作例を示しておきますので参考にしてください．

操作例5-2-2　ktocの操作方法

```
A>cc ktoc -e2100 ←ktocのコンパイルはカナ対応に変更したBDS C，あるいはα-Cで行なう．
BD Software C Compiler v1.50a (part I)
  32K elbowroom
BD Software C Compiler v1.50 (part II)
  29K to spare

A>clink ktoc
BD Software C Linker   v1.50
Linkage complete
  38K left over


A>cc ktest1 -p
BD Software C Compiler v1.50a (part I)

   1: main()
   2: {
   3:   printf ("表現力¥n");  } ←一見正しくコンパイルされるかの
   4:   printf ("十人¥n");    }   ように見えるが，
   5: }

  36K elbowroom
BD Software C Compiler v1.50 (part II)
  32K to spare

A>clink ktest1
BD Software C Linker   v1.50
Linkage complete
  43K left over

A>ktest1
阜ｯ力 } ←実行すると文字化けしてしまう．
署1   }
```

```
A>ktoc ktest1 ←――上記のktest1.cをktocに通してみる.
<F:KTEST1.CK>を作成しました

A>type ktest1.ck
main()
{
        printf ("¥225¥134¥214¥273¥227¥315¥n");  } 漢字を含まない
        printf ("¥217¥134¥2201¥n");            } 形式に変換される.
}

A>cc ktest1.ck
BD Software C Compiler v1.50a (part I)
  36K elbowroom
BD Software C Compiler v1.50 (part II)
  32K to spare

A>clink ktest1
BD Software C Linker   v1.50
Linkage complete
  43K left over

A>ktest1
表現力 }
十人   } 正しく表示される.


A>type ktest2.c

#define MSG1    "マクロ定義も使えます.¥n"
#define MSG2    "ｶﾀｶﾅ ﾓ 併用 ﾃﾞｷﾏｽ¥n"

main()
{
        printf (MSG1);
        printf (MSG2);
        printf ("表現力¥n");
        printf ("十人¥n");
}

A>cc ktest2 -p
BD Software C Compiler v1.50a (part I)
```

```
  1:
  4: main()
  5: {
  6:   printf (");
  7:   printf (MSG2);
  8:   printf ("表現力¥n");
  9:   printf ("十人¥n");
 10: }

  6: String too long (or missing quote)◄──── ktest2.cは
                                              コンパイルすら不能.

A>ktoc ktest2◄──── ktocに通してからコンパイル.
<F:KTEST2.CK>を作成しました

A>cc ktest2.ck
BD Software C Compiler v1.50a (part I)
  35K elbowroom
BD Software C Compiler v1.50 (part II)
  32K to spare

A>clink ktest2
BD Software C Linker   v1.50
Linkage complete
  43K left over

A>ktest2
マクロ定義も使えます.  ┐
ｶﾀｶﾅ ﾓ 併用 ﾃﾞｷﾏｽ     │ 制限なしに
表現力                │ すべての表示が可能.
十人                  ┘

A>
```

5-3 日本語処理パッケージ

さて，前々項，前項で，BDS C，α-Cにより，日本語処理を行なう用意は整いました．

しかし，漢字などのシフトJISコードは基本的に漢数字の8ビットのコード体系とコンパチビリティを持たせた形にしてあるものの．C言語では日本語が標準ではないために，日本語文字列の処理を行なう関数のすべては有効に使うことができません．

そのため，BDS C, α-C用に専用の日本語処理パッケージを作成しました．この関数パッケージを用いることで，日本語文字列の処理が簡単に行なうことができるようになります．是非，有効に使っていただきたいと思います．

まず，この日本語処理パッケージに含まれる関数について説明します．

```
kgetline(buf,cnt)
  char  *buf;
  int   cnt;
```

bufに内容を格納したいバッファーのアドレス，cntに入力可能な最大文字数＋1をセットするとコンソールから1行入力し，その結果をbutから格納します．戻り値は入力した文字数になります．ただし，リターンキーは文字列および文字数には含まれません．

この関数で注意することは1バイト系の文字を入力している場合には最大cnt-2文字までしか入力が行なわれないことです（終端文字の0を除く）．もし，cnt-2文字めが漢字など2バイト系文字の1バイトめであった場合には，2バイトめもコンソールから入力してバッファに格納します．

kgetline（buf, 11）の場合

　　カナ，英数字のみ　　　　　：9文字まで
　　第9文字が漢字の1バイトめ：10文字まで入力

getline, getsなどは文字列を入力する場合の基本的な関数ですが，漢字などを入力する場合にはうまくいかない場合があります．それは2バイト系の

文字が存在するためにgetlineの文字数制限により，無情にも第1バイトのみで入力が打ち切られてしまう場合があることと，BS，DELなどのキーを併用した際に漢字の第2バイトだけしか消去されず，2回キーを押さないと正しく文字を消去することができないことです．特にDEL，BSなどのキーについての問題は，通信などでも面倒な部分です．

kgetlineではDELキー（18H)，バックスペース（7FH）で前の1文字を削除する処理を行ないます．漢字などの場合は，正しく2バイト分消去します．また．コントロールXによる入力行のキャンセルもサポートしています．なお，これらの処理ではバックスペース（¥b）およびカーソルキャラクタを用いてBDOSコールでカーソル移動を行なっていますので，CP/MのBIOSの作成法およびBDS Cのバージョンによって若干動作が異なる可能性があります．また，TABキー（コントロールI，09H）の入力があった場合はDEL，コントロールXなどのキーが正しく動作しません．注意してください．

【注意】 TABの問題についてはBDOSコールのファンクション1（1文字入力)，2（1文字出力)，9（文字列出力）では実現できません．6（コンソールへの直接入出力）か，BIOSコールを用いて変更する必要があります．）

動作を確認したものは次のものです．

BDS C　vl.50a および α-C vl.51
turboCP/M v2.2（漢字版）　（シャープ）

```
atokstr(inbuf,outbuf)
  char *inbuf,*outbuf;
```

inbufで与えられる文字列のうち，1バイト文字をすべて2バイト文字に変更してoutbufに書き込みます．従って，outbufの長さはinbufの文字列の長さを終端文字0も含めてnとすると，最大2＊（n－1）＋1となります．元

々２バイトの文字は変更しません．なお，スペース（20H）はスペース２個にします．

また，ｶﾞ，ﾊﾟなど半角文字では２バイトで１文字になるものも正しく変換されます．ただし，その変換は機械的なものなので，“ｱﾟ”などとありえない文字を書くと誤って変換されます．関数atok, iskanjiを使用しています．

```
int atok(c)
  char  c;
```

１バイト文字cを２バイト文字に変換し，整数で返します．変換できない場合には０を返します．

```
ktoastr(inbuf,outbuf)
  char  *inbuf,*outbuf;
```

inbufで与えられる文字列で２バイト文字のうち１バイト文字にできるものだけを変更してoutbufに格納します．ただし，スペースが２個あっても１個には変換しませんので，atokstrで変換した文字列をktoastrに入力した場合，スペースについてのみ，異なる可能性があります．

なお，関数ktoaを用いています．

```
char ktoa(k)
  unsigned  k;
```

２バイト文字kを１バイト文字に変換します．変換できない場合は０を返します．

関数iskanji, _ksearch（文字列の検索関数）を用いています．

```
romanstr(str,buf)
  char *str,*buf;
```

ローマ字カナ変換を行ないます．この関数は英大文字あるいは小文字で与えられるローマ字の文字列をダイナミックに1バイト文字のカナに変換します．このローマ字変換の規則は次のとおりですが，この規則に当てはまらない場合，ローマ字はそのまま出力バッファにコピーされます．

	□a	□i	□u	□e	□o	
k	ｶ	ｷ	ｸ	ｹ	ｺ	
s	ｻ	ｼ	ｽ	ｾ	ｿ	
t	ﾀ	ﾁ	ﾂ	ﾃ	ﾄ	
n	ﾅ	ﾆ	ﾇ	ﾈ	ﾉ	
h	ﾊ	ﾋ	ﾌ	ﾍ	ﾎ	
m	ﾏ	ﾐ	ﾑ	ﾒ	ﾓ	
r	ﾗ	ﾘ	ﾙ	ﾚ	ﾛ	（lでも変換可能）
y	ﾔ	×	ﾕ	×	ﾖ	
w	ﾜ	ｳｨ	ｳｩ	ｳｪ	ｦ	
g	ｶﾞ	ｷﾞ	ｸﾞ	ｹﾞ	ｺﾞ	
z	ｻﾞ	ｼﾞ	ｽﾞ	ｾﾞ	ｿﾞ	
d	ﾀﾞ	ﾁﾞ	ﾂﾞ	ﾃﾞ	ﾄﾞ	
b	ﾊﾞ	ﾋﾞ	ﾌﾞ	ﾍﾞ	ﾎﾞ	
p	ﾊﾟ	ﾋﾟ	ﾌﾟ	ﾍﾟ	ﾎﾟ	
f	ﾌｧ	ﾌｨ	ﾌ	ﾌｪ	ﾌｫ	
v	ｳﾞｧ	ｳﾞｨ	ｳﾞ	ｳﾞｪ	ｳﾞｫ	
ky	ｷｬ	ｷｨ	ｷｭ	ｷｪ	ｷｮ	
sy	ｼｬ	ｼｨ	ｼｭ	ｼｪ	ｼｮ	
ty	ﾁｬ	ﾁｨ	ﾁｭ	ﾁｪ	ﾁｮ	

n y	ニャ	ニィ	ニュ	ニェ	ニョ
h y	ヒャ	ヒィ	ヒュ	ヒェ	ヒョ
m y	ミャ	ミィ	ミュ	ミェ	ミョ
r y	リャ	リィ	リュ	リョ	リョ
g y	キ゛ャ	キ゛ィ	キ゛ュ	キ゛ェ	キ゛ョ
z y	シ゛ャ	シ゛ィ	シ゛ュ	シ゛ェ	シ゛ョ
d y	チ゛ャ	チ゛ィ	チ゛ュ	チ゛ェ	チ゛ョ
b y	ヒ゛ャ	ヒ゛ィ	ヒ゛ュ	ヒ゛ェ	ヒ゛ョ
p y	ヒ゜ャ	ヒ゜ィ	ヒ゜ュ	ヒ゜ェ	ヒ゜ョ
j	シ゛ャ	シ゛	シ゛ュ	シ゛ェ	シ゛ョ
t h	テャ	ティ	テュ	テェ	テョ
d h	テ゛ャ	テ゛ィ	テ゛ュ	テ゛ェ	テ゛ョ
t s	ツァ	ツィ	ツ	ツェ	ツォ

nのみ－ン

xのみ－ン

（ウンヨウなどはuxyouとすると正しく変換されます.）

```
htokstr(str,buf)
  char *str,*buf;
```

2バイト文字のひらがなの文字列を2バイト文字のカタカナに変換します. 入力文字列の先頭アドレスをstrに，出力文字列の出力はbufに設定します. 2バイトのひらがな以外の文字はそのままコピーします.

```
unsigned htokana(k)
  unsigned k;
```

kが2バイト文字のひらがなの場合，それを2バイトのカタカナに直します．もしひらがなでない場合はそのままkの値を返します．

```
ktohstr(str,buf)
  char *str,*buf;
```

2バイト文字のカタカナ文字列をひらかなに直します．カタカナ以外の文字が含まれていた場合にはそのまま出力にコピーします．

入力はstr，出力はbufです．

```
unsigned kanatoh(k)
  unsigned k;
```

2バイト文字kがカタカナであった場合にはひらがなに直して戻り値とします．もし，カタカナでなければそのまま返します．

```
ktype(k)
  unsigned k;
```

2バイト文字kの種類を調べ，その種類によって次のような値を返します．

2バイト文字でない場合………－1
未定義…………………………… 0
記号……………………………… 1
数字……………………………… 2
英大文字………………………… 3
英小文字………………………… 4

ひらがな……………………………5

カタカナ……………………………6

漢字…………………………………7

その他(ギリシャ文字など)……8

```
iskanji(c)
  char c;
```

iskanji( )は文字cが2バイト文字の1バイトめであるかどうかを判定し，そうであれば1，さもなければ0を返します．

これらの関数を含むリストnihongo.cをリスト5-3-1に紹介します．なお，nihongo.cはカナ対応に変換したBDS C，あるいはα-Cでktocを用いてコンパイルしてください．

リスト5-3-1 nihongo.c

```
 1:  /*************************************************
 2:
 3:         BDS C日本語処理パッケージ v1.0
 4:
 5:         Author: Tsu.Mitarai  13/Dec/1986
 6:               (c) Armat co. 1986
 7:
 8:              /*   利用方法   */
 9:
10:         1.23行の KTEST を 0 に変更
11:         2."nihongo.c" をコンパイル
12:
13:                 A>ktoc nihongo
14:                 A>cc nihongo.ck
15:
16:         3.ターゲットプログラムとリンク
17:
18:                 A>cc target
19:                 A>clink target -f nihongo
20:
21:  *************************************************/
22:  #include <bdscio.h>
23:  #define KTEST   1
24:
25:  #if KTEST
26:  main()
27:  {
28:         char    buf[33];
29:         char    knbuf[33];
30:         char    kbuf[65];
```

```
 31:            char    khbuf[65];
 32:            char    kkbuf[65];
 33:            char    abuf[33];
 34:            int     n;
 35:
 36:            while (1)
 37:            {
 38:                    printf ("input->");
 39:                    n=kgetline (buf,30);
 40:                    printf ("オリジナル : %s (文字数%d)¥n",buf,n);
 41:
 42:                    romanstr (buf,kbuf);
 43:                    printf ("ｶﾅ変換      : %s¥n",kbuf);
 44:
 45:                    atokstr (kbuf,knbuf);
 46:                    printf ("ｶﾅ->カナ    : %s¥n",knbuf);
 47:
 48:                    ktohstr (knbuf,khbuf);
 49:                    printf ("カナ->かな : %s¥n",khbuf,n);
 50:
 51:                    htokstr (khbuf,kkbuf);
 52:                    printf ("かな->カナ : %s¥n",kkbuf);
 53:
 54:                    ktoastr (kkbuf,abuf);
 55:                    printf ("カナ->ｶﾅ    : %s¥n¥n",abuf);
 56:            }
 57:    }
 58:    #endif
 59:
 60:
 61:    kgetline(buf,cnt)
 62:            char    *buf;           /* output buffer */
 63:            int     cnt;            /* 最大文字数    */
 64:    {
 65:            int     ccnt;
 66:            char    c;
 67:            int     i;
 68:            char    *pnt;
 69:
 70:            pnt = buf;
 71:            ccnt = 0;
 72:            while (1)
 73:            {
 74:                    c=bdos (1);
 75:                    switch (c)
 76:                    {
 77:                    case '¥r':      /* リターンキー */
 78:                                    *pnt=0;
 79:                                    bdos (9,"¥r¥n$");
 80:                                    return(ccnt);
 81:
 82:                    case 3:         /* control C    */
 83:                                    exit ();
 84:
 85:                    case 0x7f:      /* BS,LEFT ARROW,DEL */
 86:                    case 0x8:       if (ccnt==0)
 87:                                    {
 88:                                            bdos (2,0x1c);
 89:                                            break;
 90:                                    }
 91:                                    else if (iskanji(*(pnt-2)))
 92:                                    {
 93:                                            ccnt -= 2;
 94:                                            pnt -= 2;
 95:                                            bdos (9,"¥b  ¥b¥b$");
 96:                                    }
 97:                                    else if (*(pnt-1) < 0x20)
 98:                                    {
 99:                                            bdos (9,"¥b  ¥b¥b$");
100:                                            ccnt--;
```

```
101:                                    pnt--;
102:                            }
103:                            else
104:                            {
105:                                    bdos (9," ¥b$");
106:                                    ccnt--;
107:                                    pnt--;
108:                            }
109:
110:                            break;
111:
112:                    case 0x18:      /* contol X     */
113:                            pnt = buf;
114:                            for (i=0; i<ccnt; i++)
115:                            {
116:                                    if (*pnt++<0x20) bdos (9,"¥b  ¥b¥b¥b$");
117:                                    else             bdos (9," ¥b¥b$");
118:                            }
119:                            bdos (9," ¥b$");
120:                            ccnt = 0;
121:                            pnt = buf;
122:                            break;
123:
124:                    default:        *pnt++ = c;
125:                            ccnt++;
126:                            if (c<0x20)
127:                            {
128:                                    bdos (2,'^');
129:                                    bdos (2,c+0x40);
130:                            }
131:                            if (ccnt >= cnt-2)
132:                            {
133:                                    if (iskanji(c))
134:                                    {
135:                                            *pnt++ = bdos (1);
136:                                            ccnt++;
137:                                    }
138:                                    *pnt = 0;
139:                                    bdos (9,"¥r¥n$");
140:                                    return(ccnt);
141:                            }
142:                    }
143:            }
144:    }
145:
146:
147:    /* 1 バイト文字列 -> 2 バイト文字列変換 */
148:    atokstr(str,buf)
149:            char    *str;           /* input  */
150:            char    *buf;           /* output */
151:    {
152:            unsigned k;
153:            while (*str)
154:            {
155:                    if (*str == 'ｳ' && *(str+1) == 'ﾞ')
156:                    {
157:                            *buf++ = 0x83;
158:                            *buf++ = 0x94;
159:                            str ++;
160:                    }
161:                    else if (*str == 'ﾞ')    *(buf-1)+=1;
162:                    else if (*str == 'ﾟ')    *(buf-1)+=2;
163:                    else if (iskanji(*str))
164:                    {
165:                            *buf++=*str++;
166:                            *buf++=*str;
167:                    }
168:                    else if (k=atok(*str))
169:                    {
170:                            *buf++=k/256;
```

```
171:                            *buf++=k&0xff;
172:                    }
173:                    else    *buf++=*str;
174:                    str++;
175:            }
176:            *buf=0;
177:    }
178:
179:
180:    /* 1 バイト文字 -> 2 バイト文字変換 */
181:    unsigned atok(c)
182:            char    c;
183:    {
184:            char    *s;
185:            if (c<' ')      return (0);
186:            else if (c<0x40)
187:            {
188:                    s="　！”＃＄％＆’（）＊＋，－．／０１２３４５６７８９：；＜＝＞？";
189:                    c=(c-0x20)*2;
190:            }
191:            else if (c<0x60)
192:            {
193:                    s="＠ＡＢＣＤＥＦＧＨＩＪＫＬＭＮＯＰＱＲＳＴＵＶＷＸＹＺ［￥］＾＿";
194:                    c=(c-0x40)*2;
195:            }
196:            else if (c<0x80)
197:            {
198:                    s="｀ａｂｃｄｅｆｇｈｉｊｋｌｍｎｏｐｑｒｓｔｕｖｗｘｙｚ｛｜｝～　";
199:                    c=(c-0x60)*2;
200:            }
201:            else if (c<0xa0)        return (0);
202:            else if (c<0xc0)
203:            {
204:                    s="　。「」、・ヲァィゥェォャュョッーアイウエオカキクケコサシスセソ";
205:                    c=(c-0xa0)*2;
206:            }
207:            else if (c<0xe0)
208:            {
209:                    s="タチツテトナニヌネノハヒフヘホマミムメモヤユヨラリルレロワン゛゜";
210:                    c=(c-0xc0)*2;
211:            }
212:            else    return (0);
213:            return (*(s+c)*256 + *(s+c+1));
214:    }
215:
216:
217:    /* 2 バイト文字列 -> 1 バイト文字列変換 */
218:    ktoastr(str,buf)
219:            char    *str;   /* input  */
220:            char    *buf;   /* output */
221:    {
222:            unsigned c;
223:            while (*str)
224:            {
225:                    if (c = ktoa (*str*256+*(str+1),1))
226:                    {
227:                            if (c<256)      *buf++ = c;
228:                            else
229:                            {
230:                                    *buf++ = c & 0xff;
231:                                    *buf++ = c / 256;
232:                            }
233:                            str+=2;
234:                    }
235:                    else if (iskanji (*str))
236:                    {
237:                            *buf++ = *str++;
238:                            *buf++ = *str++;
239:                    }
240:                    else
```

```
241:                    {
242:                            *buf++ = *str++;
243:                    }
244:            }
245:            *buf=0;
246:    }
247:
248:
249:    /* 2 バイト文字 -> 1 バイト文字変換
250:        濁音，半濁音の場合には上位バイトに'ﾞ'，'ﾟ'
251:        のコードが設定されます．                    */
252:
253:    unsigned ktoa(kn)
254:            unsigned kn;
255:    {
256:            char    *s;
257:            unsigned k;
258:
259:            if (!iskanji(kn/256))   return (0);
260:
261:            s="！”＃＄％＆’（）＊＋，－．／０１２３４５６７８９：；＜＝＞？";
262:            if (ERROR != (k=_ksearch (s,kn)))
263:                    return (k+0x21);
264:
265:            s="＠ＡＢＣＤＥＦＧＨＩＪＫＬＭＮＯＰＱＲＳＴＵＶＷＸＹＺ［￥］＾＿";
266:            if (ERROR != (k=_ksearch (s,kn)))
267:                    return (k+0x40);
268:
269:            s="｀ａｂｃｄｅｆｇｈｉｊｋｌｍｎｏｐｑｒｓｔｕｖｗｘｙｚ｛｜｝～　";
270:            if (ERROR != (k=_ksearch (s,kn)))
271:                    return (k+0x60);
272:
273:            s="。「」、・ヲァィゥェォャュョッーアイウエオカキクケコサシスセソ";
274:            if (ERROR != (k=_ksearch (s,kn)))
275:                    return (k+0xa1);
276:
277:            s="タチツテトナニヌネノハヒフヘホマミムメモヤユヨラリルレロワン゛゜";
278:            if (ERROR != (k=_ksearch (s,kn)))
279:                    return (k+0xc0);
280:
281:            s="ガギグゲゴザジズゼゾダヂヅデド";
282:            if (ERROR != (k=_ksearch (s,kn)))       return ('ﾞ'*256+k+0xb6);
283:
284:            s="バビブベボ";
285:            if (ERROR != (k=_ksearch (s,kn)))       return ('ﾞ'*256+k+0xca);
286:
287:            s="パピプペポ";
288:            if (ERROR != (k=_ksearch (s,kn)))       return ('ﾟ'*256+k+0xca);
289:
290:            if (k=='ヴ')                            return ('ﾞｳ');
291:
292:            return (0);
293:
294:    }
295:
296:
297:    /* 2 バイト文字列のサーチ */
298:    _ksearch(str,k)
299:            char    *str;
300:            unsigned k;
301:    {
302:            char    c0,c1;
303:            int     i;
304:
305:            i = 0;
306:            c0 = k/256;
307:            c1 = k&0xff;
308:            while (1)
309:            {
310:                    if ((*str == c0) && (*(str+1) == c1))   return (i);
```

```
311:                    if (!*(++str))  return (-1);
312:                    if (!*(++str))  return (-1);
313:                    i++;
314:            }
315:    }
316:
317:
318:    /* ローマ字文字列->ｶﾅ文字列変換 */
319:    romanstr(str,buf)
320:            char    *str;   /* input */
321:            char    *buf;   /* output */
322:    {
323:            char    kbuf[5];
324:            *buf=0;
325:            while (*str)
326:            {
327:                    str=romatok (str,kbuf);
328:                    strcat (buf,kbuf);
329:            }
330:    }
331:
332:
333:    /* ローマ字 ｶﾅ(1ﾊﾞｲﾄﾓｼﾞ) 変換 */
334:    romatok(pnt,buf)
335:            char    *pnt;
336:            char    *buf;
337:    {
338:            char    c;
339:            char    c1,c2,c3;
340:            int     ku;
341:            char    *str;
342:
343:            str = pnt;
344:            c1 = toupper (*str++);
345:
346:            if (c=_ckboin (c1))     *buf++ = c;     /* 母音 */
347:            else if (c1 == 'X')     *buf++='ﾝ';
348:            else
349:            {
350:                    c2 = toupper (*str++);
351:                    if (c=_ckboin (c2))
352:                    {
353:                            switch (c1)
354:                            {
355:                            case 'K':       *buf++=c+5;
356:                                            break;
357:                            case 'S':       *buf++=c+10;
358:                                            break;
359:                            case 'T':       *buf++=c+15;
360:                                            break;
361:                            case 'N':       *buf++=c+20;
362:                                            break;
363:                            case 'H':       *buf++=c+25;
364:                                            break;
365:                            case 'M':       *buf++=c+30;
366:                                            break;
367:                            case 'L':
368:                            case 'R':       *buf++=c+38;
369:                                            break;
370:                            case 'G':       *buf++=c+5;
371:                                            *buf++='ﾞ';
372:                                            break;
373:                            case 'Z':       *buf++=c+10;
374:                                            *buf++='ﾞ';
375:                                            break;
376:                            case 'D':       *buf++=c+15;
377:                                            *buf++='ﾞ';
378:                                            break;
379:                            case 'B':       *buf++=c+25;
380:                                            *buf++='ﾞ';
```

```
381:                            break;
382:            case 'P':       *buf++=c+25;
383:                            *buf++='ﾟ';
384:                            break;
385:            case 'Y':       if (c=='ｱ')         *buf++='ﾔ';
386:                            else if (c=='ｳ') *buf++='ﾕ';
387:                            else if (c=='ｵ') *buf++='ﾖ';
388:                            else                goto kerror;
389:                            break;
390:            case 'W':       if (c=='ｱ')         *buf++='ﾜ';
391:                            else if (c=='ｵ') *buf++='ｦ';
392:                            else
393:                            {
394:                                    *buf++='ｳ';
395:                                    *buf++=c-10;
396:                            }
397:                            break;
398:            case 'F':       if (c=='ｳ')         *buf++='ﾌ';
399:                            else
400:                            {
401:                                    *buf++='ﾌ';
402:                                    *buf++=c-10;
403:                            }
404:                            break;
405:            case 'J':       *buf++='ｼ';
406:                            *buf++='ﾞ';
407:                            c=_ckyoon (c);
408:                            if (c!='ｨ')     *buf++=c;
409:                            break;
410:            case 'V':       *buf++='ｳ';
411:                            *buf++='ﾞ';
412:                            *buf++=c-10;
413:                            break;
414:            default:        goto kerror;
415:            }
416:    }
417:    else if (c1 != 'N' && c1==c2)
418:    {
419:            *buf++='ｯ';
420:            str--;
421:    }
422:    else if (c1 == 'N' && (c1==c2 || c2==0 || c2!='Y'))
423:    {
424:            *buf++='ﾝ';
425:            str--;
426:    }
427:    else
428:    {
429:            c3 = toupper (*str++);
430:            if (!(c=_ckboin(c3)))   goto kerror;
431:
432:            c = _ckyoon (c);
433:            ku=c2*256+c1;
434:            switch (ku)
435:            {
436:            case 'KY':      *buf++='ｷ';
437:                            *buf++=c;
438:                            break;
439:            case 'SY':      *buf++='ｼ';
440:                            *buf++=c;
441:                            break;
442:            case 'TY':      *buf++='ﾁ';
443:                            *buf++=c;
444:                            break;
445:            case 'NY':      *buf++='ﾆ';
446:                            *buf++=c;
447:                            break;
448:            case 'HY':      *buf++='ﾋ';
449:                            *buf++=c;
450:                            break;
```

```
451:                            case 'MY':      *buf++='ﾐ';
452:                                            *buf++=c;
453:                                            break;
454:                            case 'RY':      *buf++='ﾘ';
455:                                            *buf++=c;
456:                                            break;
457:                            case 'GY':      *buf++='ｷ';
458:                                            *buf++='ﾞ';
459:                                            *buf++=c;
460:                                            break;
461:                            case 'ZY':      *buf++='ｼ';
462:                                            *buf++='ﾞ';
463:                                            *buf++=c;
464:                                            break;
465:                            case 'DY':      *buf++='ﾁ';
466:                                            *buf++='ﾞ';
467:                                            *buf++=c;
468:                                            break;
469:                            case 'BY':      *buf++='ﾋ';
470:                                            *buf++='ﾞ';
471:                                            *buf++=c;
472:                                            break;
473:                            case 'PY':      *buf++='ﾋ';
474:                                            *buf++='ﾟ';
475:                                            *buf++=c;
476:                                            break;
477:                            case 'SH':      *buf++='ｼ';
478:                                            if (c!='ｨ')      *buf++=c;
479:                                            break;
480:                            case 'CH':      *buf++='ﾁ';
481:                                            if (c!='ｨ')      *buf++=c;
482:                                            break;
483:                            case 'TS':      *buf++='ﾂ';
484:                                            if (c!='ｭ')      *buf++=c;
485:                                            break;
486:                            case 'TH':      *buf++='ﾃ';
487:                                            *buf++=c;
488:                                            break;
489:                            case 'DH':      *buf++='ﾃ';
490:                                            *buf++='ﾞ';
491:                                            *buf++=c;
492:                                            break;
493:                            default:        if (cl=='N')
494:                                            {
495:                                                    *buf++='ﾝ';
496:                                                    str -= 2;
497:                                            }
498:                                            else    goto kerror;
499:                            }
500:                    }
501:            }
502:            *buf=0;
503:            return (str);
504:
505:    kerror: *buf++=*pnt++;
506:            *buf=0;
507:            return (pnt);
508:    }
509:
510:
511:    /* 母音のチェック */
512:    _ckboin(c)
513:            char    c;
514:    {
515:            switch (c)
516:            {
517:            case 'A':       return ('ｱ');
518:            case 'I':       return ('ｲ');
519:            case 'U':       return ('ｳ');
520:            case 'E':       return ('ｴ');
```

```
521:            case 'O':        return ('ｵ');
522:            default:;        return (0);
523:            }
524:    }
525:
526:
527:    /* よう音のチェック */
528:    _ckyoon(c)
529:            char    c;       /* ｶﾅ */
530:    {
531:            switch (c)
532:            {
533:            case 'ｱ':        return ('ｧ');
534:            case 'ｲ':        return ('ｨ');
535:            case 'ｳ':        return ('ｭ');
536:            case 'ｴ':        return ('ｪ');
537:            case 'ｵ':        return ('ｮ');
538:            default:;
539:            }
540:            return (c);
541:    }
542:
543:
544:    /* ひらがな文字列 ->カタカナ文字列変換 */
545:    htokstr(str,buf)
546:            char    *str;
547:            char    *buf;
548:    {
549:            unsigned k;
550:            while (1)
551:            {
552:                    if (!*str)              break;
553:                    if (!iskanji (*str))    *buf++=*str++;
554:                    else
555:                    {
556:                            k=*str++;
557:                            if (!*str)              break;
558:                            k=k*256+*str++;
559:                            k=htokana (k);
560:                            *buf++=k/256;
561:                            *buf++=k & 0xff;
562:                    }
563:            }
564:            *buf=0;
565:    }
566:
567:
568:    /* ひらがな->カタカナ変換 */
569:    unsigned htokana(k)
570:            unsigned k;
571:    {
572:            if (ktype (k) == 5)
573:            {
574:                    if (k>0x82dd)   return (k+0xa2);
575:                    else            return (k+0xa1);
576:            }
577:            return (k);
578:    }
579:
580:    /* カタカナ文字列 ->ひらがな文字列変換 */
581:    ktohstr(str,buf)
582:            char    *str;
583:            char    *buf;
584:    {
585:            unsigned k;
586:            while (1)
587:            {
588:                    if (!*str)              break;
589:                    if (!iskanji (*str))    *buf++=*str++;
590:                    else
```

```
591:                    {
592:                            k=*str++;
593:                            if (!*str)              break;
594:                            k=k*256+*str++;
595:                            k=kanatoh (k);
596:                            *buf++=k/256;
597:                            *buf++=k & 0xff;
598:                    }
599:            }
600:            *buf=0;
601:    }
602:
603:
604:    /* カタカナ->ひらがな変換 */
605:    unsigned kanatoh(k)
606:            unsigned k;
607:    {
608:            if (ktype (k) == 6)
609:            {
610:                    if (k>0x837f)   return (k-0xa2);
611:                    else            return (k-0xa1);
612:            }
613:            return (k);
614:    }
615:
616:
617:    /* 2バイト文字の種類をチェックする関数
618:       出力: -1 (2バイト文字ではない)
619:              0 (未定義)
620:              1 (記号)
621:              2 (数字)
622:              3 (英大文字)
623:              4 (英小文字)
624:              5 (ひらがな)
625:              6 (カタカナ)
626:              7 (漢字)
627:              8 (ギリシャ文字など)          */
628:
629:    ktype(k)
630:            unsigned k;
631:    {
632:            char    kl;
633:            kl = k&0xff;
634:
635:            if (!iskanji (k/256))   return (-1);
636:            if (kl<0x40)            return (0);
637:            if (kl==0x7f)           return (0);
638:            if (kl>0xfc)            return (0);
639:
640:            if (k<0x8140)           return (0);
641:            if (k<0x824f)           return (1);
642:            if (k<0x8258)           return (2);
643:            if (k<0x8260)           return (0);
644:            if (k<0x827a)           return (3);
645:            if (k<0x8281)           return (0);
646:            if (k<0x829a)           return (4);
647:            if (k<0x829f)           return (0);
648:            if (k<0x82f2)           return (5);
649:            if (k<0x8340)           return (0);
650:            if (k<0x8397)           return (6);
651:            if (k<0x849f)           return (8);
652:            return (7);
653:    }
654:
655:
656:    /* 2バイト文字列かどうかのチェック */
657:    iskanji(c)
658:            char    c;
659:    {
660:            if (c<0x80)     return (FALSE);
```

```
661:            if (c<0xa0)     return (TRUE);
662:            if (c<0xe0)     return (FALSE);
663:            if (c<0xfd)     return (TRUE);
664:            return (FALSE);
665:    }
666:
```

5-4 日本語処理パッケージの使い方

nihongo.cはそのままではテスト用の関数を含んでいますので23行めの#define文のktestを0に変更してコンパイルし，nihongo.crlというファイルにしておきます．これを用いてプログラムを作成する場合には，

```
clink program -f nihongo
```

という形で使うと自動的にnihongo.crlから必要関数だけをピックアップして使うことが可能です．

また，BDS Cであらかじめ用意されている文字列処理用の関数のうち，日本語をうかつに使うと問題になるものがあります．toupper, tolowerがその代表的なもので，この関数を用いて文字列などを変換すると漢字コードの2バイトめが英大文字あるいは小文字と同じコードのものはすべて異なった文字に変換されてしまいます．そのためあらかじめ2バイトコードでないことを確認して使う必要があります．

なお，ktocの処理は決して速いとは言えません．そこで，日本語の文字列がある部分だけを別のファイルにしておき，そこだけktocで変換しても良いでしょう．日本語の無い部分までktocに通す必要はありません．

操作例5-4-1 日本語処理パッケージの操作例

```
A>ktoc nihongo ←――日本語処理パッケージのコンパイルにはktocが必要.
<NIHONGO.CK>を作成しました

A>cc nihongo.ck ←――コンパイルにも漢字対応に変更したものを使う.
BD Software C Compiler v1.50a (part I)
  21K elbowroom
```

```
BD Software C Compiler v1.50 (part II)
  23K to spare

A>clink nihongo
BD Software C Linker   v1.50
Linkage complete
  36K left over

A>nihongo←――――テストプログラムの実行

input->akasatanahamayarawan
オリジナル : akasatanahamayarawan (文字数20)
ｶﾅ変換     : ｱｶｻﾀﾅﾊﾏﾔﾗﾜﾝ←――――――――ローマ字 → カナ変換
ｶﾅ->カナ   : アカサタナハマヤラワン←――半角カナ → 全角カナ
カナ->かな : あかさたなはまやらわん←――全角カナ → ひらがな
かな->カナ : アカサタナハマヤラワン←――ひらがな → 全角カナ
カナ->ｶﾅ   : ｱｶｻﾀﾅﾊﾏﾔﾗﾜﾝ←――――――――全角カナ → 半角カナ

input->fikyunyahyuryajathidhitsu
オリジナル : fikyunyahyuryajathidhitsu (文字数25)
ｶﾅ変換     : ﾌｨｷｭﾆｬﾋｭﾘｬｼﾞｬﾃｨﾃﾞｨﾂ←――――拗音も正しく変換できる.
ｶﾅ->カナ   : フィキュニャヒュリャジャティディツ
カナ->かな : ふぃきゅにゃひゅりゃじゃてぃでぃつ
かな->カナ : フィキュニャヒュリャジャティディツ
カナ->ｶﾅ   : ﾌｨｷｭﾆｬﾋｭﾘｬｼﾞｬﾃｨﾃﾞｨﾂ

input->ro-ma字 ｶﾀｶﾅ カタカナ
オリジナル : ro-ma字 ｶﾀｶﾅ カタカナ (文字数21)
ｶﾅ変換     : ﾛｰﾏ字 ｶﾀｶﾅ カタカナ←――――ローマ字以外の部分はそのまま.
ｶﾅ->カナ   : ローマ字 カタカナ カタカナ
カナ->かな : ろーま字 かたかな かたかな
かな->カナ : ローマ字 カタカナ カタカナ
カナ->ｶﾅ   : ﾛｰﾏ字 ｶﾀｶﾅ ｶﾀｶﾅ

input->漢字 ﾓ 問題 nashi.
オリジナル : 漢字 ﾓ 問題 nashi. (文字数18)
ｶﾅ変換     : 漢字 ﾓ 問題 ﾅｼ.
ｶﾅ->カナ   : 漢字 モ 問題 ナシ.
カナ->かな : 漢字 も 問題 なし.
かな->カナ : 漢字 モ 問題 ナシ.
カナ->ｶﾅ   : 漢字 ﾓ 問題 ﾅｼ.
```

```
input->^C
A>
```

以上，日本語処理についてかなり詳細に述べました．BDS $，α-Cで日本語処理はできないと思っていた方々には参考にしていただけると思います．また，最低限これだけでも機能が揃うと新しいプログラムの可能性も生れてきます．是非，新分野に挑戦していただきたいと思います．

第 6 章
オーバーレイとチェイン

BDS C, α-Cによって，本格的なプログラムを作成してくるようになると，やがて問題になってくることがあります．

その1つはCP/M(および8ビットCPUの）最大の欠点である．メモリサイズの問題で，プログラムが大きくなり過ぎたため，プログラムがすべてメモリに入り切らなくなった場合です．これには本質的な手だてはありません．あるとすれば，CPUを十分なメモリを確保できる16ビット以上のものに交換することでしょう．

勿論これは現実的ではありませんから，実際にはプログラムをいくつかに分割し，必要とする際にそのプログラムをディスクからロードするオーバーレイの技法が良く用いられます．本章ではBDS C特有のテクニックであるオーバーレイとチェインについて解説します．

6-1 プログラムのチェイン

プログラムが大きくなり，どうしてもメモリ上にすべてが入りきらなくなってきた場合，ディスクからプログラムを呼び出しながら実行するオーバーレイ技法を必要とするようになります．これは非常に大きなプログラムでなくとも，作業用のワークメモリエリアを大きく確保したい場合にも用いられることがあります．

実行中のプログラムがその一部を切り放して別のプログラムモジュールをディスクからロードして実行を継続するオーバーレイに対し，実行中のプログラムすべてを捨て去り，全く新しいプログラムをディスクからロードして実行させることをチェインと言います（BASICではMERGE，およびRUN "filename" に当るでしょうか.）．

BDS Cでは，オーバーレイは比較的簡単にできるように設計されており，いくつかの手順を踏むだけで実現できます．ただし，プログラムをディスクから呼び出しながら実行していくため，プログラムを機能単位で分割し，あらかじめ整理することがどうしても必要です．単純にプログラムが大きくなり過ぎたから，というだけではオーバーレイ構造をとることはできません．しかし，別個の実行ファイル(comファイル)を作るチェインで構わないのであれば，もっと楽に作成することができます．BDS C, α-Cにはexec, execv, excelなどチェイン用の関数が用意されていますので，作成は簡単です．

この方法は，プログラムのランタイムパッケージの部分がダブってロードされるため，ディスク容量，ファイルロード時間などの点で不利ですが，全く別のプログラムとしてデバッグできるため，非常に簡便で，いつでも実現できる方法だと言えます．

BDS Cの場合指定しない限り外部変数はすべてクリアされてしまいますので，メモリ上のデータを渡すことはできませんが，execv, execlの場合はコマンド行からの引数を渡すこともできますから最低限の情報を送ることは可能です．

リスト6-1-1 execv関数によるチェイン

```
A>type execroot.c

#include <bdscio.h>

main(argc,argv)
        int     argc;
        char    *argv[];
{
        printf ("Root segment excuting.¥n");
        argv[argc]=0;
        execv ("execovl.com",&argv[1]);
}
```

呼び出し側のプログラム.

呼び出される側のプログラム.
これはリスト2-2-1のargs.cとほとんど同じもの.
単体で実行できる.

```
A>type execovl.c

#include <bdscio.h>

main(argc,argv)
        int     argc;
        char    *argv[];
{
        int     i;
        printf ("Execovl excuting...¥n");
        for (i=1; i<argc; i++)
        {
                printf ("arg%d=¥"%s¥"¥n",i,argv[i]);
        }
}

A>cc execroot
BD Software C Compiler v1.50a (part I)
  35K elbowroom
BD Software C Compiler v1.50 (part II)
  32K to spare

A>clink execroot
BD Software C Linker   v1.50
Linkage complete
```

```
  42K left over


A>cc execovl
BD Software C Compiler v1.50a (part I)
  35K elbowroom
BD Software C Compiler v1.50 (part II)
  32K to spare

A>clink execovl
BD Software C Linker   v1.50
Linkage complete
  43K left over


A>execroot a bc def ghij
Root segment excuting.
Execovl excuting....  ←――― execovlが起動された.
arg1="A"   ┐
arg2="BC"  │
arg3="DEF" ├ execrootから正しく文字列が渡されている.
arg4="GHIJ"┘

A>
```

なお，呼び出される側のプログラムのリンク時に－zオプション（外部変数クリア禁止）を指定し，かつ呼び出し側のプログラムと外部変数の順番，先頭アドレスを揃えておくとすべての変数を受け渡すことが可能です.ただし，BDS C付属のＬ２リンカー（α-Cにはありません）では－zに相当するオプションがないので，生成されたCOMファイルをDDTなどでパッチを当て，禁止するしか方法はありません（141番地をＣ３Ｈ→Ｃ９Ｈに変更する）.

また，execv, execlなどの関数では，呼び出すのはCP/MのユーティリティであるPIPやASMあるいはCコンパイラでも構わないという利点もあります（但し，DIR, ERAなどのビルトインコマンドは使えません.）.

BDS Cのコンパイラ本体であるcc. comもこのexecv, execlなどからコマンドラインの文字列を設定して実行させることが可能であり，そうすると

ktocのようなプリプロセッサも利用が簡単になります.

そこで，ktocのリストのmain関数だけを変更して自動的にコンパイラを起動するck.comというプログラムに作り換えてみましょう．ckに対するオプションはそのままcc.comに対するオプションとなりますので, nk（カタカナのみ）オプションはなくなっています.

リスト6-1-2 ck.cと操作例

```
main(argc, argv)
int     argc;
char    *argv[];
{
        char    infile[20];
        char    outfile[20];

        if ( argc < 2 )
        {
                printf ("書式: ck filename [options]¥n");◄────────変更
                exit ();
        }
        strcpy (infile,argv[1]);
        strcpy (outfile,argv[1]);
        strcat (infile,".C");
        strcat (outfile,".CK");

        kanjif = TRUE;

        if (ERROR == fopen (infile,fpin))
        {
                printf ("<%s>がオープンできません",infile);
                exit ();
        }
        if (ERROR == fcreat (outfile,fpout))
        {
                printf ("<%s>を作成することができません",outfile);
                exit ();
        }
        ktoc ();                        /* メインルーチン */
        fputc (CPMEOF);

        if (ERROR == fclose (fpout))
                printf ("<%s>を正しくクローズできません",outfile);
        else
        {                                                  ┐
                printf ("<%s>を作成しました¥n",outfile);   │
                argv[1] = outfile;                         ├ 変更
                argv[argc]=0;                              │
                execv ("cc.com",&argv[1]);                 │
        }                                                  ┘
}
                (以降変更なし)
```

```
A>cc ck -e2200 ←—— execv関数が加わるためサイズが大きくなる．－e2200でコンパイル．
BD Software C Compiler v1.50a (part I)
  32K elbowroom
BD Software C Compiler v1.50 (part II)
  29K to spare


A>clink ck
BD Software C Linker   v1.50
Linkage complete
  38K left over


A>ck ck -e2200 -p ←—— 完成したck.comで自分自身をコンパイルしてみる．
<CK.CK>を作成しました
BD Software C Compiler v1.50a (part I) ←—— ck.comにより自動的に
                                          コンパイルが起動される．
          ⋮(前略)
  24: main(argc, argv)
  25: int       argc;
  26: char      *argv[];
  27: {
  28:   char    infile[20];
  29:   char    outfile[20];
  30:
  31:   if ( argc < 2 )           ←—— このように変換されているため，
  32:   {                              入力ファイルは正しくck.ckである．
  33:           printf ("¥217¥221¥216¥256: ck filename [options]¥n");
  34:           exit ();
  35:   }
          ⋮(中略)
 195:                   printf ("¥225¥266¥216¥232¥227¥361¥202¥252¥222¥267¥211
¥337¥202¥254¥202¥351¥202¥251¥"¥203}¥201¦¥203N¥202¥252¥227¥216¥202¥277¥202¥304
¥202¥242¥202¥334¥202¥267");
 196:                   exit ();
 197:           }
 198:           else    fputc (*pnt++);
 199:   }
 200: }

  31K elbowroom
BD Software C Compiler v1.50 (part II)
  29K to spare

A>clink ck
BD Software C Linker   v1.50
Linkage complete
  38K left over
A>
```

なお，このckを利用する場合，cc.comは必ずデフォルトドライブ（A>の

時にはドライブA）に入れておく必要があります．

また，main関数の最後で

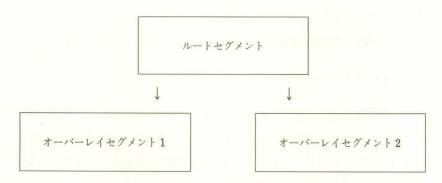
```
argv[argc]=0;
```

という式がありますが，ポインタ配列argvは0からargc-1までの要素しか存在していないので，本来このような使い方をしてはいけません．しかし，BDS C，α-Cではランタイムパッケージの部分にあらかじめ30個までargv用の空間が確保されており，問題が発生することはありません．

6-2　swapin関数を用いたオーバーレイ

さて，プログラムが長くなり，実行時に必要なプログラムモジュールをロードする本格的なオーバーレイをとらねばならなくなった場合，あらかじめプログラムを次のような観点から，機能ごとに振り分ける必要がでてきます．

1．常に存在しなければならない関数（機能）
2．必要に応じて実行できれば良い関数（機能）

1．はオーバーレイする際の常駐プログラム（ルートセグメント）となり，2がオーバーレイセグメントになります．このオーバーレイセグメントを複数用意し，どのプログラムをロードするかによって実行できる機能を切り換えます．

図でお判りのように，オーバーレイセグメント1が存在しているときにはオーバーレイセグメント2の関数は実行することはできません．両者共通に必要な関数はルートセグメントに入れておくか，両方のファイルに同じように入れておかなければなりません．

また，これはコンパイルしてから言えることですが，ルートセグメントのサイズそのものが大きくなり過ぎると共用メモリ領域が狭くなり，オーバーレイセグメントが入り切らなくなります．特に，1～2Kバイトしかない空間をいくつものオーバーレイセグメントで共用するという方法は余りにも煩雑であり，うまいやり方とは言えないでしょう．このような場合には関数のレイアウトを考慮して4Kバイト程度のメモリをまず確保し，その上で，オーバーレイセグメントは2つなり3つなりのファイルに厳選するほうが良いと思います．この時，オーバーレイセグメント1と2が大体同じようなサイズになると，効率も良くなります．

オーバーレイセグメントの読み込みはルートセグメントから行ないます．この時，このファイル名をovlprg1.comとし，読み込むアドレスを0x4000とすると，

```
swapin("ovlprg1.com",0x4000);
```

としてメモリ上にロードする方法が良く用いられます．0x4000はそのままオーバーレイセグメントを呼び出す時のアドレスでもあります． swapin関数

は簡単にディスク上のファイルの内容をメモリに呼び出すことができるので便利ですが，ファイルの長さをチェックしないので注意しなければならない場合もあります．

オーバーレイ関数の実行は関数へのポインタを用いて間接的に行ないます．

```
int (*ovlprg) ();
        ⋮
ovlprg=0x4000;
(*ovlprg) (引数) ;
```

という形になります．上記の方法はオーバーレイ時に限らず，関数名ではなくアドレスで関数呼出しを実行する場合の一般的な方法です．

それでは実際の例をご紹介します．

リスト6-2-1　オーバーレイのサンプルと操作例

```
A>type overlay.c

#include <bdscio.h>

main()
{
        char    buf[4];
        while (1)
        {
                printf ("Which function (1 or 2)..");
                getline (buf,3);
                if      (*buf=='1')     ovl1 ();
                else if (*buf=='2')     ovl2 ();
        }
}


ovl1()
{
```

```
        int     (*ovlprg) ();
        if (ERROR == swapin ("ovl1.com",0x4000))◀─┐
                printf ("ovl1 overlay error !¥n");│
        else                                  └─swapinにより
        {                                       オーバーレイセグメント
                                                を読み込む.
                ovlprg = 0x4000;
                (*ovlprg) ();◀────オーバーレイセグメントの実行.
        }
}


ovl2()
{
        int     (*ovlprg) ();
        if (ERROR == swapin ("ovl2.com",0x4000))
                printf ("ovl2 overlay error !¥n");
        else
        {
                ovlprg = 0x4000;
                (*ovlprg) ();
        }
}


A>type ovl1.c

#include <bdscio.h>

main()
{
        printf ("it is in ovl1¥n");
}


A>type ovl2.c

#include <bdscio.h>

main()
{
```

```
        printf ("it is in ovl2¥n");
}

A>cc overlay -e6000
BD Software C Compiler v1.50a (part I)
  34K elbowroom
BD Software C Compiler v1.50 (part II)
  31K to spare

A>clink overlay -w
BD Software C Linker   v1.50
Linkage complete
  42K left over

A>cc ovl1 -e6000
BD Software C Compiler v1.50a (part I)
  35K elbowroom
BD Software C Compiler v1.50 (part II)
  32K to spare

A>clink ovl1 -v -14000 -y overlay
BD Software C Linker   v1.50
Linkage complete
  46K left over

A>cc ovl2 -e6000
BD Software C Compiler v1.50a (part I)
  35K elbowroom
BD Software C Compiler v1.50 (part II)
  32K to spare

A>clink ovl2 -v -14000 -y overlay
BD Software C Linker   v1.50
Linkage complete
  46K left over


A>overlay
```

(A>cc overlay -e6000 への注記) オーバーレイ時には外部変数とコードがダブらないように外部変数アドレスを指定する.

(A>clink ovl1 -v -14000 -y overlay への注記)
－v　オーバーレイセグメント指定.
－l4000　実行開始アドレスを4000Hに指定.
－y overlay　overlay.symというシンボルファイルを読込み，既に存在する関数はリンクしない.

```
Which function (1 or 2)..1
it is in ovl1←―――――オーバーレイセグメントをロードして実行.
Which function (1 or 2)..2
it is in ovl2
Which function (1 or 2)..1
it is in ovl1
Which function (1 or 2)..2
it is in ovl2
Which function (1 or 2)..^C
A>
```

リスト6-2-1は小さい2つのオーバーレイセグメントを4000H番地から読み込み，それと同時に実行させている例です．1を選ぶとovl1.comを，2を選ぶとovl2.comを読み込みます．実行はロードアドレスと同じく4000H番地からです．

ここで重要なことはこのオーバーレイセグメントの中にはメインプログラム中に存在するprintfなどの関数が含まれておらず，オーバーレイセグメントからのprintf関数の呼出しはメインプログラム内のサブルーチンをコールすることで行なっているということです．これらの処理は自動的に行なわれますので，頭を悩ませることはありません．

6-3　オーバーレイ関数の呼び出し方法

さて，前項の例はオーバーレイモジュール1つにつき1つの関数だけでしたが，実際にこのような場合に問題になるのは次の2点です．

1．モジュールが既にロードされているかどうかの確認をどうするか．
2．モジュールの中に複数の関数がある場合，どうやって呼び出しを行なうか．

実行前に必ずファイルをロードしてやれば問題はありませんが，オーバーレ

イ時はかなりディスクアクセスに時間がかかりますから，もし既に必要なセグメントがメモリ上に存在しているなら，ロードせずに済ませたいものです．また，ルートセグメント（常駐のメインプログラム）からはオーバーレイセグメントの先頭アドレス以外は知ることができませんので，関数が2つ以上ある場合，2番め以降の関数をダイレクトに呼び出すことができません．

そこで，手続は少し面倒になりますが，オーバーレイセグメントの1番最初に，あらかじめチェック用の関数を記述しておき，現在存在しているモジュールの番号を返すようにしておきます．すると，必要な関数を呼び出す際に，まずメモリ上にあるかどうかをテストして，なければディスクからロード，あればそのまま関数を呼び出せば良いわけです．また，この関数を引数により複数の関数を呼び出す複合関数にしておけば，呼び出しアドレスが判らなくとも構わないことになります．

具体的には次のように行ないます．

リスト6-3-1　オーバーレイのチェック

```
A>type overlay2.c

#include <bdscio.h>

#define FUNC0   0
#define FUNC1   1

#define funca() func0(1)
#define funcb() func0(2)
#define funcc() func1(1)
#define funcd() func1(2)

main()
{
        char    buf[4];
        if (ERROR == swapin ("func0.com",0x4000))
        {
                printf ("No overlay segment.");
                exit ();
        }
        while (1)
```

関数funca～funcdをマクロ定義している．このようにすると引数の異なる同一関数の呼び出しが異なる関数の呼び出しのように扱える．

起動時に何か1つモジュールを読み込んでおくことが必要．

```
        {
                printf ("Which function (1-4)..");
                getline (buf,3);
                if      (*buf=='1')     funca ();
                else if (*buf=='2')     funcb ();
                else if (*buf=='3')     funcc ();
                else if (*buf=='4')     funcd ();
        }
}

func0(n)
        int     n;
{
        int     (*ovlprg) ();
        ovlprg = 0x4000;
        if ( (*ovlprg)(0) != FUNC0)
        {
                if (ERROR == swapin ("func0.com",0x4000))
                {
                        printf ("func0 overlay error !¥n");
                        return;
                }
        }
        (*ovlprg)(n);
}

func1(n)
        int     n;
{
        int     (*ovlprg) ();
        ovlprg = 0x4000;
        if ( (*ovlprg)(0) != FUNC1)
        {
                if (ERROR == swapin ("func1.com",0x4000))
                {
                        printf ("func1 overlay error !¥n");
                        return;
                }
        }
        (*ovlprg)(n);
}


A>type func0.c
```

メインルーチンではオーバーレイを意識せず，通常の関数呼び出しのように扱える．

メモリ上にあるセグメントがfunc0.comでなければファイルをロードする．

引数を持ってオーバーレイセグメント上の関数を実行する．

オーバーレイセグメント0

```
#include <bdscio.h>

#define FUNC0   0

main(n)
        char    n;
{
        switch (n)
        {
        case 0: return (FUNC0);←──── オーバーレイセグメント種類のチェック．当然，ファイル毎に戻り値は変える必要がある．
        case 1: funca ();
                return;
        case 2: funcb ();
                return;

        default:;
        }
}

funca()←──── 実際に実行したい関数funca
{
        printf ("This is function A¥n");
}

funcb()
{
        printf ("function B excuting.¥n");
}


A>type func1.c

#include <bdscio.h>

#define FUNC1   1

main(n)
        char    n;
{
        switch (n)
        {
        case 0: return (FUNC1);
        case 1: funcc ();
                return;
        case 2: funcd ();
                return;
```

```
        default:;
        }
}

funcc()
{
        printf ("This is function C¥n");
}

funcd()
{
        printf ("function B excuting.¥n");
}
```

〈ルートセグメントのコンパイル〉

```
A>cc overlay2 -e6000
BD Software C Compiler v1.50a (part I)
  34K elbowroom
BD Software C Compiler v1.50 (part II)
  31K to spare

A>clink overlay2 -w
BD Software C Linker   v1.50
Linkage complete
  42K left over
```

〈オーバーレイセグメント0のコンパイル〉

```
A>cc func0 -e6000
BD Software C Compiler v1.50a (part I)
  35K elbowroom
BD Software C Compiler v1.50 (part II)
  32K to spare

A>clink func0 -v -14000 -y overlay2
BD Software C Linker   v1.50
Linkage complete
  46K left over
```

〈オーバーレイセグメント1のコンパイル〉

```
A>cc func1 -e6000
BD Software C Compiler v1.50a (part I)
  35K elbowroom
BD Software C Compiler v1.50 (part II)
  32K to spare
```

```
A>clink func1 -v -l4000 -y overlay2
BD Software C Linker    v1.50
Linkage complete
  46K left over


A>overlay2

Which function (1-4)..1
This is function A                ファイル読出しは行なわれない.
Which function (1-4)..2
function B excuting.              ファイル読出しが行なわれる.
Which function (1-4)..3
This is function C                ディスクアクセスなし.
Which function (1-4)..4
function B excuting.              ディスクアクセスあり.
Which function (1-4)..2
function B excuting.              ディスクアクセスなし.
Which function (1-4)..1
This is function A                ディスクアクセスあり.
Which function (1-4)..4
function B excuting.              ディスクアクセスなし.
Which function (1-4)..3
This is function C
Which function (1-4)..^C

A>
```

リスト6-3-1のような構造にしておけば，不要な場合にはディスクアクセスをせずに済みますし，メインプログラムからもいちいち気にせずにオーバーレイ部分のプログラムを一般の関数と同じように呼び出せます．ちょっとした仮想記憶方式になるわけです．ただ，いずれにせよディスクアクセスを伴う場合が存在するわけですから，その関数の実行時間の予測がつきにくく，リアルタイムの制御には向きません．

リスト6-3-1の例ではオーバーレイのプログラムを呼び出す場合に引数が設定されておらず，複数の関数を呼び出す複合関数も簡単に作成できるのですが，数多く引数を持つ場合，および引数の数が不定の場合にはこの複合関数の作成方法自体が問題になります．一般に，引数をintなどで受け，実際

の関数呼び出し時に必要なものだけを引数に設定するようにしておけば型は異なっても値としては同じものになりますので，正しく実行されます．ただ，引数設定はもともとメモリと実行時間を浪費しますから，オーバーレイする程メモリに余裕がなくなってきた場合にはなるべく行ないたくないものです．

そのような際にはじかに関数を呼び出す方法をとることもできます．ただしこの場合，基本的にはルートセグメントのプログラムのリンク時にオーバーレイ部分のリンク終了しており，アドレスが決定していなければなりません．ところが，オーバーレイのリンク時には逆にルートセグメントのアドレスがすべて決定していなければならないのですから，どうコンパイルするかが問題になります．

これはなかなか難しいのですが，まず関数名とでためらなアドレスを記述したダミーのシンボルファイルを作成し，リンカーをごまかしてなんとかルートセグメントのCOMファイルを作ってしまいます．その際にシンボルファイルの出力を必ず指定しておき，得られたシンボルファイルを利用してオーバーレイモジュールを作成します．

その際に－ｓ，あるいは－ｗオプションにより，シンボルリストを出力し，確定したアドレスを再びダミーのシンボルファイルに書き込み，ルートセグメントをリンクします．オーバレイセグメントのリンク時に出力されるシンボルファイルにはルートセグメントのアドレスも出力されているため，使うことはできません．

操作例6-3-2　2回ルートセグメントをリンクする方法

```
A>type overlay3.c
#include <bdscio.h>

main()
{
        char    buf[4];
        if (ERROR == swapin ("func.com",0x4000))
        {
                printf ("No overlay segment.");
```

```
                exit ();
        }
        while (1)
        {
                printf ("Which function (1-4)..");
                getline (buf,3);
                if      (*buf=='1')     func1 ();
                else if (*buf=='2')     func2 ();
                else if (*buf=='3')     func3 ();
                else if (*buf=='4')     func4 ();
        }
}

A>type func.c
#include <bdscio.h>

main()  {}←────── CLINKでリンクする場合は
                  ダミーでmain関数が必要.
                  (3バイトのロスになる)

func1()
{
        printf ("function 1 excuting.¥n");
}

func2()
{
        printf ("This is func 2¥n");
}

func3()
{
        printf ("FUNCTION 3,HERE¥n");
}

func4()
{
        printf ("this is 4th function¥n");
}


A>cc overlay3 -e6000
BD Software C Compiler v1.50a (part I)
  35K elbowroom
BD Software C Compiler v1.50 (part II)
  32K to spare
```

```
A>type dummy.sym          ← あらかじめダミーの
0000      FUNC1             シンボルファイルを作成しておく．
0000      FUNC2
0000      FUNC3
0000      FUNC4
  ↑         ↑── 関数名
  └─────────── エントリアドレス

A>clink overlay3 -y dummy -w   ← dummy.sym を読み込ませ，シンボルファイル
BD Software C Linker   v1.50      出力を指定してリンク．
Linkage complete                  ここで作成された overlay3.com は実行できな
  42K left over                   いので注意．

A>cc func -e6000          ← オーバーレイセグメントのコンパイル．
BD Software C Compiler v1.50a (part I)
  35K elbowroom
BD Software C Compiler v1.50 (part II)
  32K to spare

A>clink func -v -l4000 -y overlay3 -s   ← オーバーレイセグメントのリンク．
BD Software C Linker   v1.50
Ignoring duplicate function: FUNC4 ┐ 「重複関数あり」というエラーが出力される．
Ignoring duplicate function: FUNC4 │ 多分，FUNC1～FUNC4についてのエラーだ
Ignoring duplicate function: FUNC4 │ と思われるが，FUNC4について4回出力され
Ignoring duplicate function: FUNC4 ┘ た．CLINKのバグであろう．
                                     overlay3. sym の中に同じ関数名が存在するた
                                     め，このようなエラーとなるが，func.crl 中
                                     の関数が優先されるため問題にならない．

1109 CLOSE     0FB9 EXIT      4003 FUNC1     402A FUNC2
404B FUNC3     406D FUNC4     0FBC GETLINE   0EBE ISDIGIT
0F8A ISLOWER   4000 MAIN      0FF8 OPEN      09FA PRINTF
110C PUTCHAR   1060 READ      0986 SWAPIN    0F50 TOUPPER
0EED _GV2      0A1D _SPR      0E16 _USPR
                                             ～～部のアドレス
                                             をメモしておく．
Last code address: 4093
Top of memory: DC05
Linkage complete
  46K left over

A>ed dummy.sym            ← dummy.sym を書き換える．

    : *#A
   1: *4T
   1:   0000      FUNC1  ┐
   2:   0000      FUNC2  │ 元の内容．
   3:   0000      FUNC3  │
   4:   0000      FUNC4  ┘
```

```
1: *I
1:  4003    FUNC1 ┐
2:  402A    FUNC2 │
3:  404B    FUNC3 │ シンボル出力に従ってアドレスを列挙.
4:  406D    FUNC4 ┘
5:
5: *4K
 : *B4T
1:  4003    FUNC1
2:  402A    FUNC2
3:  404B    FUNC3
4:  406D    FUNC4
1: *E


A>clink overlay3 -y dummy←────正しいdummy.symを用いて
BD Software C Linker   v1.50       ルートセグメントをもう一度リンク.
Linkage complete                   コンパイルの必要はない.
  42K left over

A>overlay3
Which function (1-4)..1 ┐
function 1 excuting.    │
Which function (1-4)..2 │
This is func 2          │ すべて正しく実行された.
Which function (1-4)..3 │
FUNCTION 3,HERE         │
Which function (1-4)..4 ┘
this is 4th function
Which function (1-4)..^C

A>
```

しかし，この方法は余りにも不便ですし，間違いやすいのでお勧めできませんが，もし用いる場合には，オーバーレイモジュールがロードされているかどうかのチェックはかならず行なってください．

なお，オーバーレイプログラムを作成する場合にはリンカーとしてclink以外にBDS C付属のL2を用いることができます．考え方は同じですが，オーバーレイセグメントのファイル中にmain関数が無くても良い点，プログラムサイズが若干小さくなる点で有利になります．

あとがき

BDS C，α-Cはコンパイル及びリンクが速いC言語です．本書の執筆にはX1turbo＋RAMディスク2基という構成で利用しましたが，短いリストはほとんど数秒でコンパイルが終了し，類似の構成で動作させた16ビット用のコンパイラよりも速いのには驚かされました．このため，ちょっとしたプログラム作成には当分手離すことができそうにありません．サブセットであるということ，ソースファイルを一旦メモリにすべて読み込んでから展開するタイプであることがこの高速性を実現しているのでしょう．

C言語の学習という観点から考えるならば，このコンパイル速度はかなり重要なポイントだと思われます．最初はほとんどがコンパイル&リンクの繰り返しですから，意欲を失わずに学習効果を上げることができると思います．

なお，本書は私が初心者にC言語を教えるためにまとめたものですが，同時に上級者のためのテクニックも解説した積りです．また，本書でわからないところはマニュアルを必ず参照してください．

さらに高度な部分を知りたい方は拙著「BDS Cプログラミング」を併せて参考にして頂くと良いでしょう．

本書が少しでも読者の方々のお役に立てば幸いです．

御手洗　毅

仕事の関係で，私もC言語を勉強しなければならなくなりました．初めはC言語になじめずとても苦労しましたが，少しずつ理解し，ようやく自分でプログラムを作ることができるようになりました．その際の学習ノートが本書の原型になっています。

本書は私のテンポにあわせて進みますからおそらく，どなたでも無理なくC言語を学んでいただけるものと思います．私が理解に苦しんだ，関数の中での引数の扱いかたや，ポインタ，構造体の使い方について詳細に解説されていると思います．

また，4章以降は内容的に上級者向きになっています．この部分は，何が書かれているかを読み取ればよいでしょう．この章から長いプログラムがでてきますが，総てを理解する必要はありません．とにかく，そのプログラムが何をするものなのかをつかめば良いと思います．最後に，私に辛抱強く教えてくれた夫(御手洗　毅)，プログラムを作成するのに時間がかかり，御迷惑をおかけした工学図書の方々に厚く御礼申し上げます．

御手洗　理英

コンパイルエラーメッセージ一覧表

BDS Cおよびα-Cでコンパイル時に発生するエラーのメッセージをＡＢＣ順にならべてあります．

〈エラーメッセージの見方〉

■インクルードファイルのとき

```
Include @ 2： 2： Missing semicolon
```

2番目にインクルードしているファイル（@の後の最初の2）

インクルードファイルの2行め（2番目の2）

■ソースファイルのとき

```
5： Missing semicolon
```

ファイルの5行め（5）

指定行でエラーが発生したことを示しますが，必らずしもプログラムの誤りはその行にあるとは限りません．

Attribute mismatch from previous declaration

異なる構造体などの中で同一のメンバ名を使う場合，その属性は同一でなければならない．

Bad arg to unary operator

単項演算子の使い方の誤り．

Bad argument list

関数の引数中におかしなものがある．

Bad array base

配列として宣言されてないのに，添字を付けている．

Bad case constant

caseの定数がおかしい．

Bad constant

定数がない．

Bad decimal digit

10進定数の中に，数字（0～9）以外の文字がある．

Bad declaration syntax

変数宣言の文法が間違っている．キーワード（char，int 等）を含む文の残りが，宣言文でない時に起こる．

Bad demension value

配列宣言で，サイズを示す添字は，定数または，定数式でなければならない．

Bad expression

式が正しくない．

Bad for syntax

間違ったfor 文がある．forの（ ）中には；が2つなければならない．

Bad function name

関数でないのに関数呼出しを行なおうとした．

Bad left operand in assignment expression

代入式での左項が代入できない定数などになっている．

Bad octal digit

0で始まる8進定数に，0～7以外の数字がある時に起こる．

Bad parameter list element

関数定義の引数パラメータの中に，間違った識別子（変数名）がある．

Bad parameter list syntax

関数定義の引数リスト中にコンマで区切られた変数名以外のものがある．

Bad structure or union member

構造体または，共有体のメンバとして，宣言されてない名前が，ピリオド，または，―>演算子の右側にある．

Bad structure or union specification

構造体または，共有体として，宣言されてない名前が，ピリオド演算子の左側にある．

Bad subscript

配列の添字がおかしい．

Bad symbols

－yオプションで指定されたシンボルファイルが，シンボルファイルの形式をしていない．

Bad type in binary operation

2進数演算の中に，おかしなタイプがある．

Bad use of member name

構造体，または，共有体のメンバーとして宣言された識別子は，構造体，または，共有体操作以外では，使うことができない．

Cannot open： 〈filename〉

〈filename〉というファイルが見付からない.

Can't close： 〈filename〉

〈filename〉で示されるファイルが，クローズできない.

Can't create CRL file

出力ドライブのディレクトリ・フル.

Can't find CC2. COM; writing CCI file to disk

CCがCC 2を見付けることができず，CCIエクステンションで中間ファイルをディスクに書いた.

Can't have more than one default：

1つのswitch文の中に，1つ以上のdefault：がある.

Close error

ファイルを，クローズしようとした時エラーが起きた.

Compilation aborted by control-C

コンパイル中に，コントロールCがタイプされたらコンパイルは中断する.

Conditional expr bad or beyond implemented subset

＃ifなどのプリプロセッサは標準C言語のサブセットである.

CRL Dir overflow：break up source file

1つのソースファイル中に関数の数が多すぎる．ソースファイルを2つに分割したほうがよい.

Curly-braces mismatched somewhere in this function

表示行からはじまる関数に{(カーリーブレイス)が多すぎる.

Declaration too complex

このエラーは，関数のレベルが多すぎたり，カッコが多すぎたりした時に起こる。

Dir full

ディスクのディレクトリエリアが，いっぱいなため，ファイルを出力できない.

Disk read error

ディスクからデータを読もうとした時，エラーが起きた.

Duplicate label

ラベルが2重に定義されている.

Encountered EOF unexpectedly (check curly-brace balance)

プログラムが，途中で終っている．これは，コメントか{が閉じてない時に起こる.

EOF found when expecting #endif

条件コンパル（#if, #ifdef, #ifndef 等）に，#endifがない.

Error on file output……disk full?

ファイルにデータを書こうとした時のエラー．ディスクに，空きエリアがない場合が多い.

Error writing : <filename>

<filename> で示されるファイルが，書き込みエラーとなった.
ディスク・フルの場合が多い.

Expecting (

普通，while，if，switchキーワードの後の（がない時に起こる。

Expecting :

：がない．

Expecting { in struct or union def

構造体または共有体宣言に{がない．

Expecting { in switch statement

switch文の{がない．switch文の式の部分は，複文を含む{ }の前になければならない．

Expecting while

do……while文で，whileがない．

Function definition not external

関数定義関数の中にある．
多くの場合，；（セミコロン）を忘れている。

Illegal break or coutinue

break文，または，continue文が間違った位置にある。

Illegal colon

おかしなコロン（：）がある．

Illegal { encountered externally

{が，おかしな所にあった時に起こる．

Illegal external statement

関数の外に，おかしな文がある．これは，普通，}が多すぎる時に起こる．

Illegal indirection

ポインターの扱いがおかしい．ポインターではないのに，ポインターとして使っている変数がある．

Illegal statement

おかしな文がある．

Illegal structure or union

構造体宣言の構造体タグ位置に現われる識別子が，以前に，構造体タグと違うもので宣言されている．

I'm totally confused. Check your control structure!

変な文字があるか，または，{ }のネストがおかしい．

#include files nested too deep

#includeのファイルが，相互に#includeしている．

Internal error : garbage in file or bug in C

CC 2で，このエラーが起きたら，コンパイラのバグと考えられる．BDソフトウェアに連絡して下さい．

Lvalve needed with ＋＋ or －－ operator

単純変数以外はインクリメント及びデクリメントできない．

Lvalue reguired

代入演算で左項がない．

Mismatched control structure

構造体の制御がおかしい．{ }がマッチしていない．

Mismatched parenthesis

カッコがマッチしてない．

Mismatched square brackets

〔 〕がマッチしてない．

Missing from formal parameter list : 〈name〉

変数の宣言が，関数名の後のパラメータリストにない．

Missing function(s) : 〈list-of-names〉

（関数名並び）

〈list-of-names〉中に示されるファイルが見つからない．

Missing { in function def.

関数定義中に { がない．プログラム中に，誤った { がない．

Missing legal identifier

必要な識別子が式の中にない．

Missing or misplaced (

(がないか，位置がおかしい．

Missing or misplaced)

) がないか，位置がおかしい．

Missing parameter list

マクロ化された＃define中の識別子が，パラメータなしに現われている．

例えば，

＃define sqr (A) (A) ＊ (A)

と定義されているのに，

d ＝sqr

の様に使われている．

Missing semicolon

；がない．しかし，；がない場合，他の無意味なエラーをひき起こす場合が多い．

Need explicit dimension size

配列のサイズが明記されてない．配列宣言でサイズを省くことができるのは，その配列が，関数の引数である時だけである．

Not in a conditional block

これは，＃endifがあるのに，＃if，＃ifdef，＃ifndefがない．

Out of symbol table space；specify more……

ＣＣのためのシンボルテーブル領域がオーバーフローした．このエラーが出たら，ソースファイルをいくつかに分けた方がよい．

Parameter mismatch

マクロ化された＃define中の識別子の，パラメータの数が合わない．

Redeclaration of：〈name〉

〈name〉で示される変数が，二重宣言されている。

Sorry；out of memory

ソースファイルが，大きすぎて，メモリーに入らない．このエラーがでたら，ソースファイルを分割した方がよい．

Sorry, out of memory. Break it up !

ファイルが，大きすぎてメモリ中に入らない．

String overflow; call BDS

#define指定で，たいへん長い識別名を多く持ったために，起こるプリプロセッサ文字列テーブルオーバーフロー．

String too long (or missing quote)

文字列が，長すぎる．（または，" がない）．

Sub-expression too deeply nested

このエラーは，永久に続く，多重の代入文について発生する．

Syntax error

はっきりしない文法的エラー．

<text> : option error

間違ったオプションが指定された．

The function <foo> is too complex; break it up a bit

関数fooが長く複雑すぎる．関数を2つに分割しなさい．

Too many cases (200 max per switch)

1つのswitch文の中のcaseが，200を越えている．

Too many dimensions

配列の次元が多すぎる．BDS Cでは，2次元まで．

Too many functions (63 max)

1つのBDS Cソースファイルは，63コの関数までしか，含むことはできない．

Undeclared identifier : <name>

<name> で示される名の変数が宣言されてない.

Undefined label used

未定義のラベルが使われている.

Unmatched left parenthesis

(が, 合わない.

Unmatched right brace

{ がないか, 意味のない } があるか.

Warning : Ignoring unknown preprocessor directive

ＢＤＳ Ｃでサポートされてないプリプロセッサ指定がある. このプリプロセッサは, 無視される.

参 考 文 献

1. BDS Cプログラミング：御手洗 毅 著
(工学図書)

2. プログラミング言語C：B.W.カーニハン/D.M.リッチー 共著
(共立出版)　　　　：石田 晴久 訳

3. BDS C Compiler User's Guide：Leor Zolman 著
(ライフボート)

4. BDS Cユーザーズマニュアル：リオー・ゾールマン 著
(ライフボート)

索　　引

≪和　文≫

≪欧　文≫

【 R 】

【 S 】

【 T 】

【 U 】

【 W 】

【 X 】

≪その他≫

BDS C/α-C関連ソフトウェアについて

(有)アーマットではBDS C/α-C関連ソフトウェアとして「BDS Cユーティリティパッケージ」を発売しています．このパッケージの中には，Z80ニーモニック用のアセンブラプリプロセッサ (zcasm), CRLファイル用の逆アセンブラ (dacrl) などが収められており，BDS C/α-Cで本格的なプログラム開発を考えている方には大変有効な内容になっています．これらのソフトウェアについての評細は，工学図書発行の「BDS Cプログラミング」をご覧ください．なお，本書で紹介した「日本語処理パッケージ」，「拡張サブミットコマンド」も含まれておりますので，本書のディスクサービスとしてもご利用ください．

現金書留か郵便振替で住所，氏名，年齢，勤務先，電話番号，使用システム，ディスクタイプを明記の上，下記宛にお申し込み下さい．販売は通信販売のみです．

ディスク内容

fplong.h	float，long関数のヘッダーファイル
zcasm.c	casmのZ80版ソースプログラム
zcasm.com	casmのZ80版実行ファイル
mrasm.com	Z80ニーモニックのアブソリュートアセンブラ(Vl.3)
dacrl.com	CRLファイルをZ80アセンブラのソースファイルに変換するユーティリティ
nihongo.c	日本語処理ユーティリティ
ktoc.c	日本語変換ユーティリティ
exsub.c (@.com)	拡張サブミットコマンド (ソースファイルと実行ファイル)
exsub2.c(/.com)	拡張サブミットコマンド (ソースファイルと実行ファイル)

その他,「BDS Cプログラミング」のディスクサービスで扱っていたものはすべて含まれます．

価　格

10,000円 (送料を含む)

対象システム

PC-8001，8801シリーズ(5インチ2D) X1，X1 turboシリーズ(5インチ2D)，CP/M標準(8インチ片面単密)，その他のシステムの方は，あらかじめお問い合わせください．

申し込み先

〒227 横浜市緑区荏田町473-5

㈲アーマット　TEL. 045-911-7427　郵便振替　横浜5 －30518

■ 著者略歴

御手洗　毅（みたらいつよし）

昭和30年 9 月24日生れ

昭和53年　慶応義塾大学工学部計測工学科卒

昭和53年　カシオ計算機㈱入社後電子楽器開発に従事.

昭和58年　㈱フジテレビジョン入社（CG 担当）

昭和61年　㈲アーマット設立

著書　「スーパーインポーズCP/M」（工学図書）

「BDS Cプログラミング」　（工学図書）

御手洗　理英（みたらいりえ）

昭和35年 3 月18日生れ

昭和56年　駿台電算専門学校卒

昭和56年　㈱フジミック入社後ソフトウェア開発に従事

昭和61年　㈲アーマット

現在　東京工業専門学校講師

著者の承
認により
検印省略

学習 BDS C・α-C

昭和62年 4 月 6 日　初　版

昭和62年 6 月30日　第 2 版

著　者　御手洗　毅・理英

発行者　小林泰明

発行所　工学図書株式会社

（営業所）東京都千代田区九段北1-14-15

〒102　電話　東京（262）3772 番

振替東京 7-13465 番

印刷所　株式会社サンヨー

定価 1,900円

ISBN4-7692-0164-8 C3058